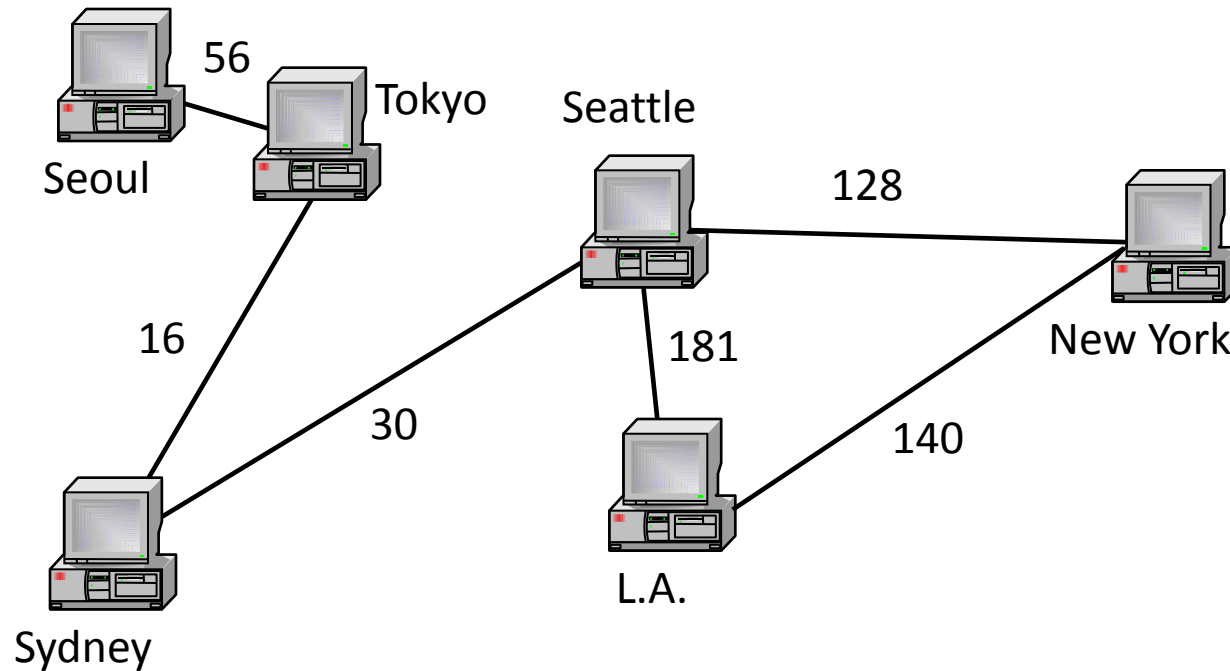
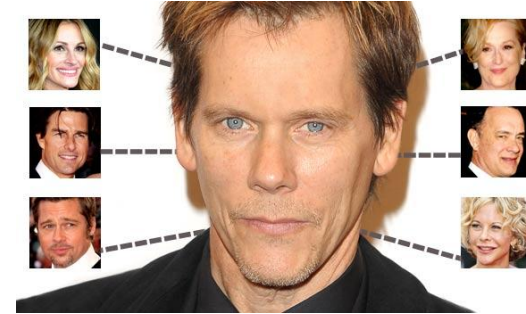
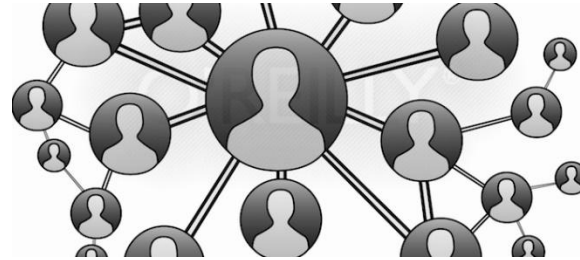
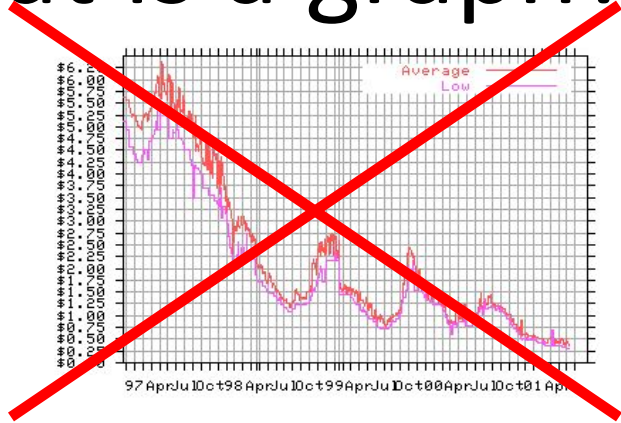


Chapter-5 : Searching Algorithms and String Matching Algorithms

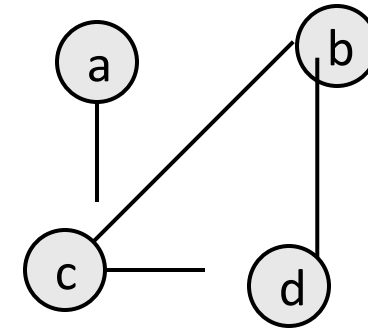
Nisha Panchal

What is a graph?



Graphs

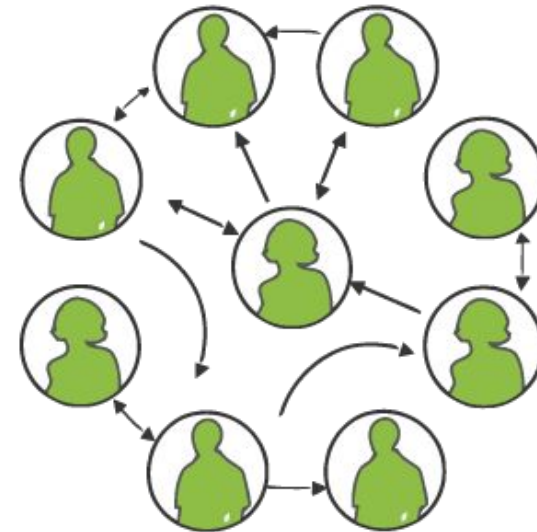
- **graph**: A data structure containing:
 - a set of **vertices** V , *(sometimes called nodes)*
 - a set of **edges** E , where an edge represents a connection between 2 vertices.
 - Graph $G = (V, E)$
 - an edge is a pair (v, w) where v, w are in V



- the graph at right:
 - $V = \{a, b, c, d\}$
 - $E = \{(a, c), (b, c), (b, d), (c, d)\}$
- **degree**: number of edges touching a given vertex.
 - at right: $a=1, b=2, c=3, d=2$

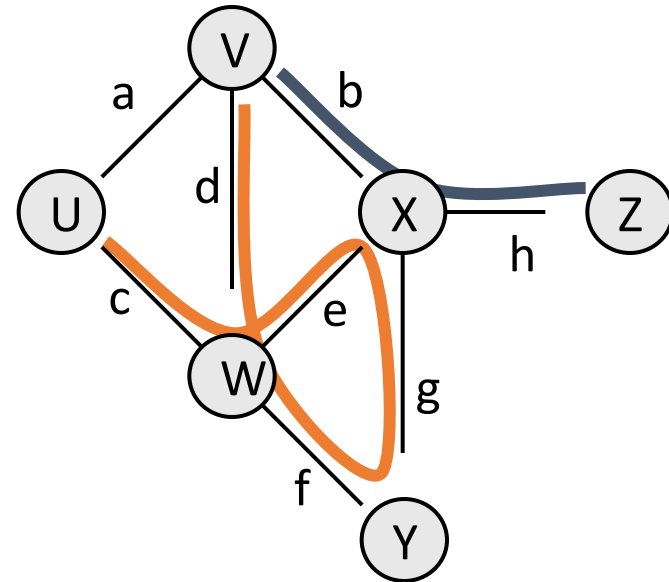
Graph examples

- For each, what are the vertices and what are the edges?
 - Web pages with links
 - Methods in a program that call each other
 - Road maps (e.g., Google maps)
 - Airline routes
 - Facebook friends
 - Course pre-requisites
 - Family trees
 - Paths through a maze



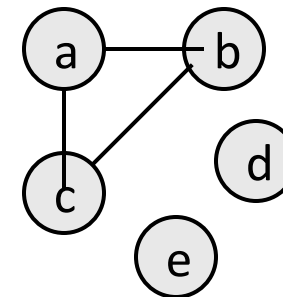
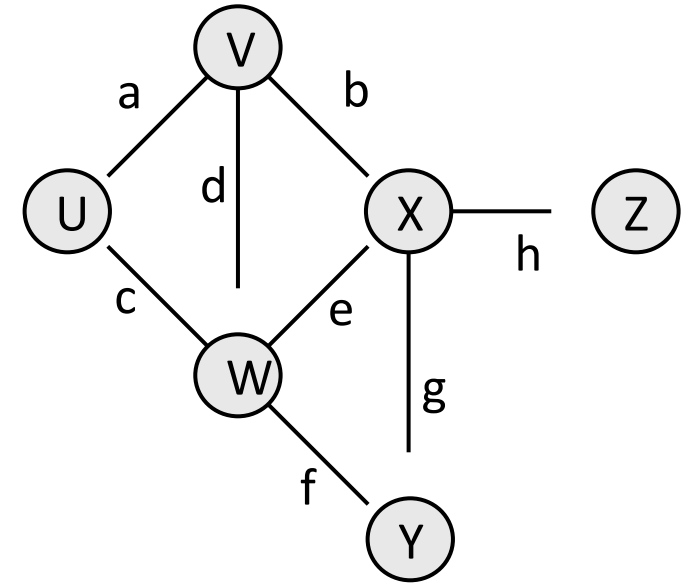
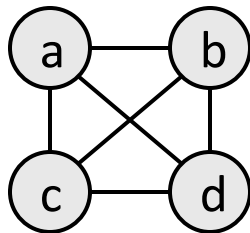
Paths

- **path:** A path from vertex a to b is a sequence of edges that can be followed starting from a to reach b .
 - can be represented as vertices visited, or edges taken
 - example, one path from V to Z : $\{b, h\}$ or $\{V, X, Z\}$
 - What are two paths from U to Y ?
- **path length:** Number of vertices or edges contained in the path.
- **neighbor** or **adjacent:** Two vertices connected directly by an edge.
 - example: V and X



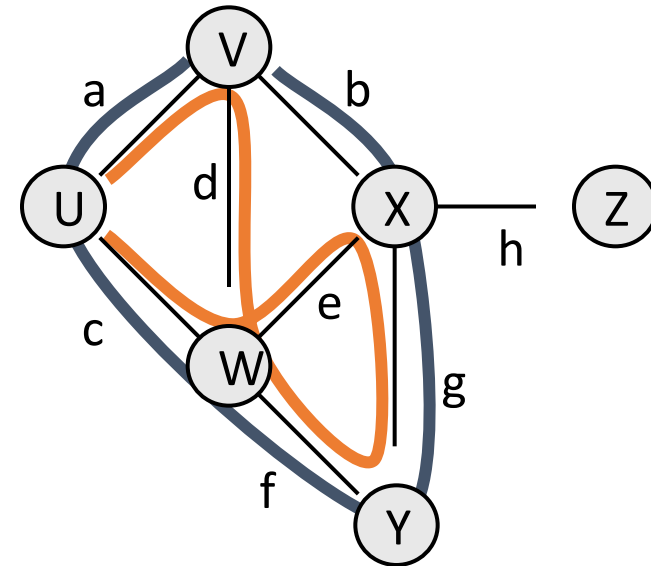
Reachability, connectedness

- **reachable:** Vertex a is *reachable* from b if a path exists from a to b .
- **connected:** A graph is *connected* if every vertex is reachable from any other.
 - Is the graph at top right connected?
- **strongly connected:** When every vertex has an edge to every other vertex.



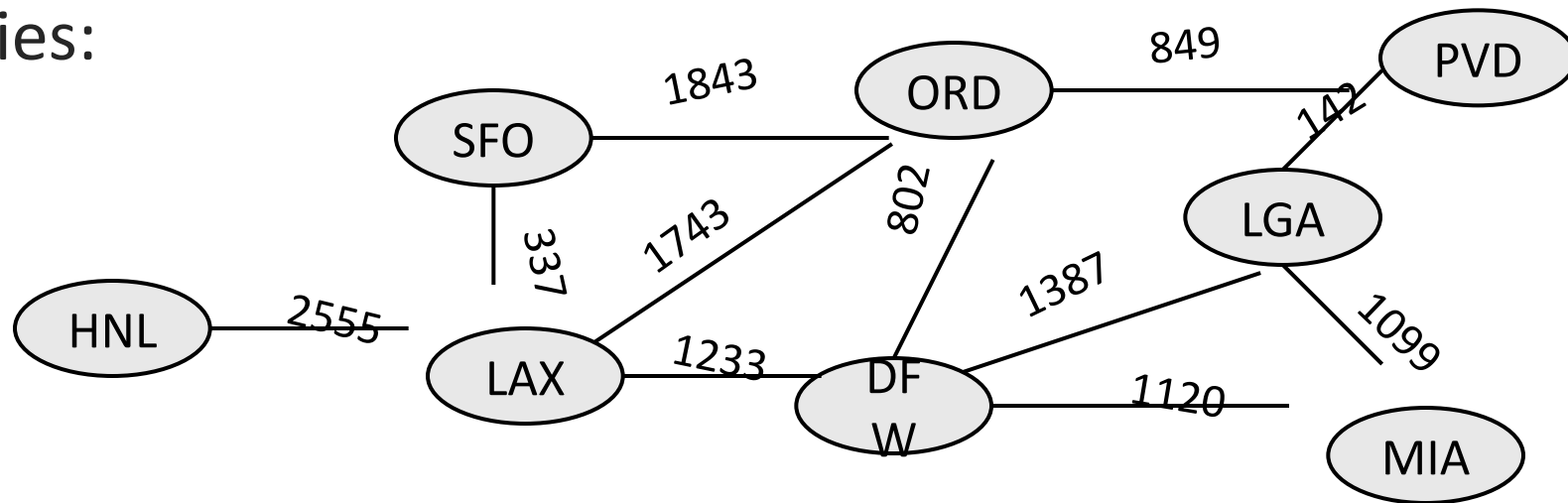
Loops and cycles

- **cycle:** A path that begins and ends at the same node.
 - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
 - example: {c, d, a} or {U, W, V, U}.
- **acyclic graph:** One that does not contain any cycles.
- **loop:** An edge directly from a node to itself.
 - Many graphs don't allow loops.



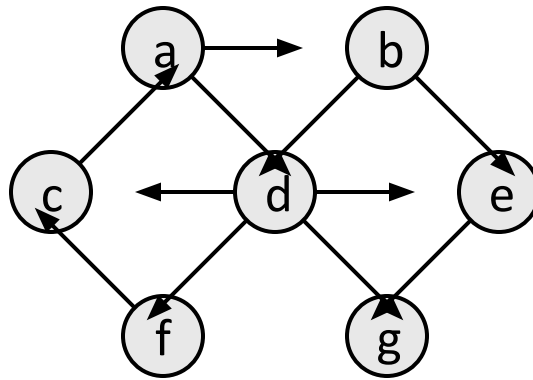
Weighted graphs

- **weight:** Cost associated with a given edge.
 - Some graphs have weighted edges, and some are unweighted.
 - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
 - Most graphs do not allow negative weights.
- *example:* graph of airline flights, weighted by miles between cities:



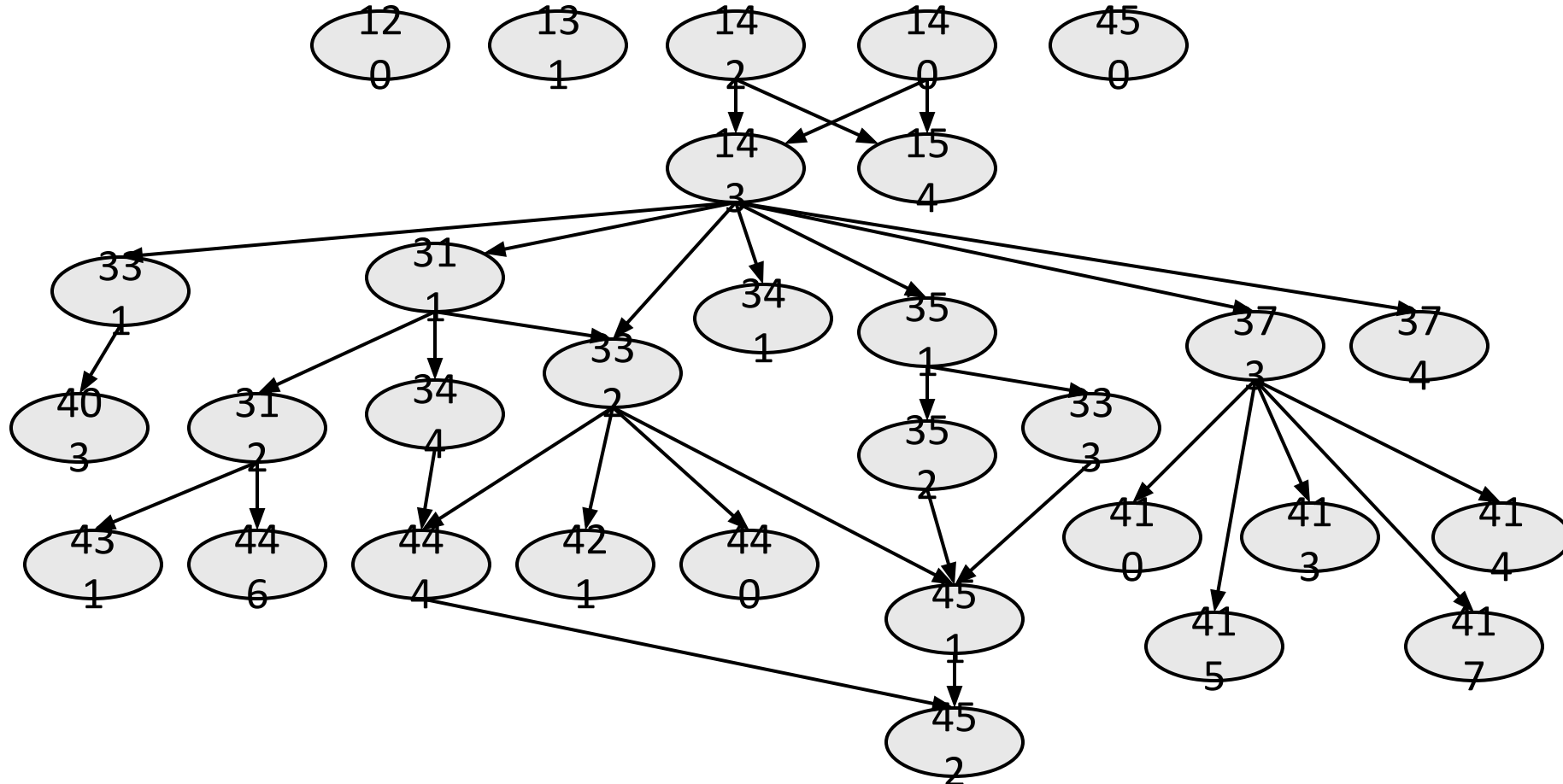
Directed graphs

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
 - If graph is directed, a vertex has a separate in/out degree.
 - A digraph can be weighted or unweighted.
 - Is the graph below connected? Why or why not?



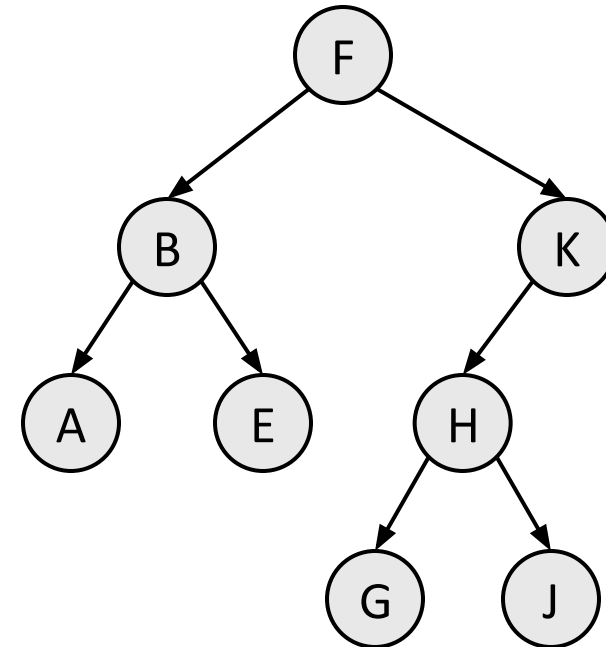
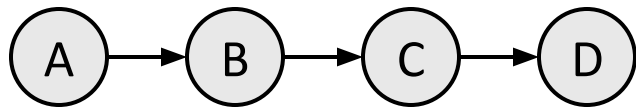
Digraph example

- Vertices = UW CSE courses (incomplete list)
- Edge (a, b) = a is a prerequisite for b



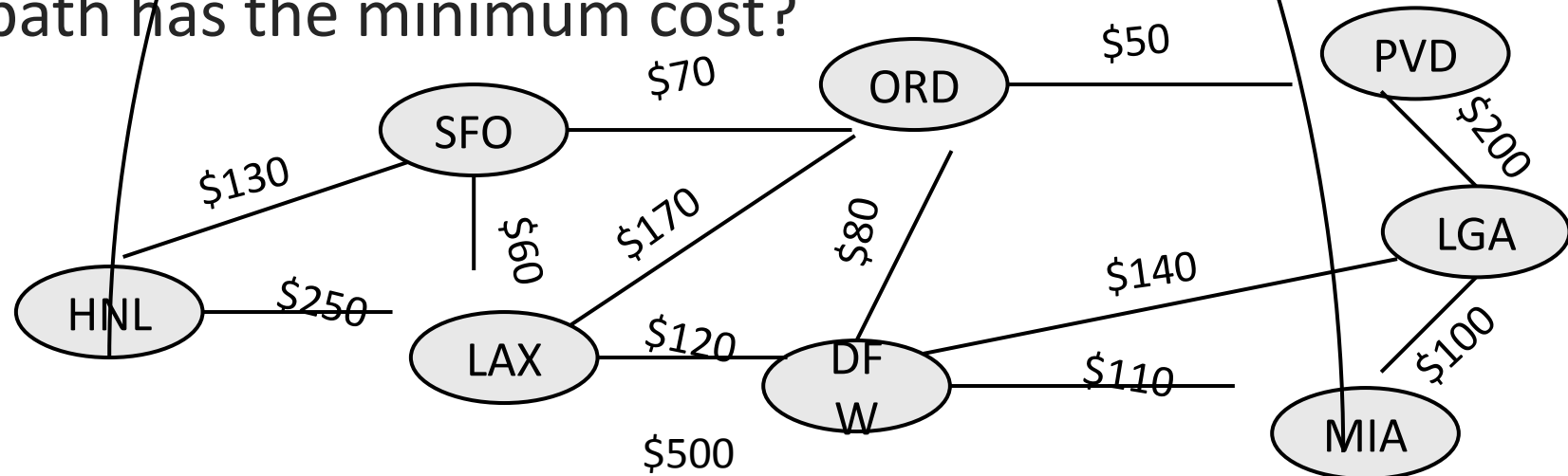
Linked Lists, Trees, Graphs

- A *binary tree* is a graph with some restrictions:
 - The tree is an unweighted, directed, acyclic graph (DAG).
 - Each node's in-degree is at most 1, and out-degree is at most 2.
 - There is exactly one path from the root to every node.
- A *linked list* is also a graph:
 - Unweighted DAG.
 - In/out degree of at most 1 for all nodes.



Searching for paths

- Searching for a path from one vertex to another:
 - Sometimes, we just want *any* path (or want to know there *is* a path).
 - Sometimes, we want to minimize path *length* (# of edges).
 - Sometimes, we want to minimize path *cost* (sum of edge weights).
- What is the shortest path from MIA to SFO?
Which path has the minimum cost?



BFS (Breadth-First Search):

- **Concept:** BFS starts at a root node and systematically explores all its neighbor nodes at the same level (breadth) before moving on to the next level. It uses a **queue** data structure to keep track of the unvisited nodes, following a "First In, First Out" (FIFO) principle.

- **Applications in Design Analysis:**

- Finding the shortest path in unweighted graphs: Since BFS explores all neighbors at a level first, it guarantees finding the shortest path between two nodes if the edge weights are all equal.
- Level order traversal: BFS can be used to visit all nodes in a tree or graph level by level, which can be helpful for layout optimization or dependency analysis.
- Network analysis: BFS is commonly used in network analysis tasks like finding connected components in a graph or identifying broadcast domains.

- **Input:** A graph G and a starting node s .
- **Output:** Visit all nodes in the graph in BFS order.
- **Data Structures:**
 - Queue: To store the unvisited nodes.
 - Visited array: To keep track of visited nodes (optional).
- **Algorithm:**
 - Initialize an empty queue and a visited array (optional, all elements set to False).
 - Mark the starting nodes as visited (if using visited array).
 - Enqueue s into the queue.
 - While the queue is not empty:
 - Dequeue a node u from the queue.
 - Process node u (e.g., print it).
 - For each unvisited neighbor v of u :
 - Mark v as visited (if using visited array).
 - Enqueue v into the queue.

DFS (Depth-First Search):

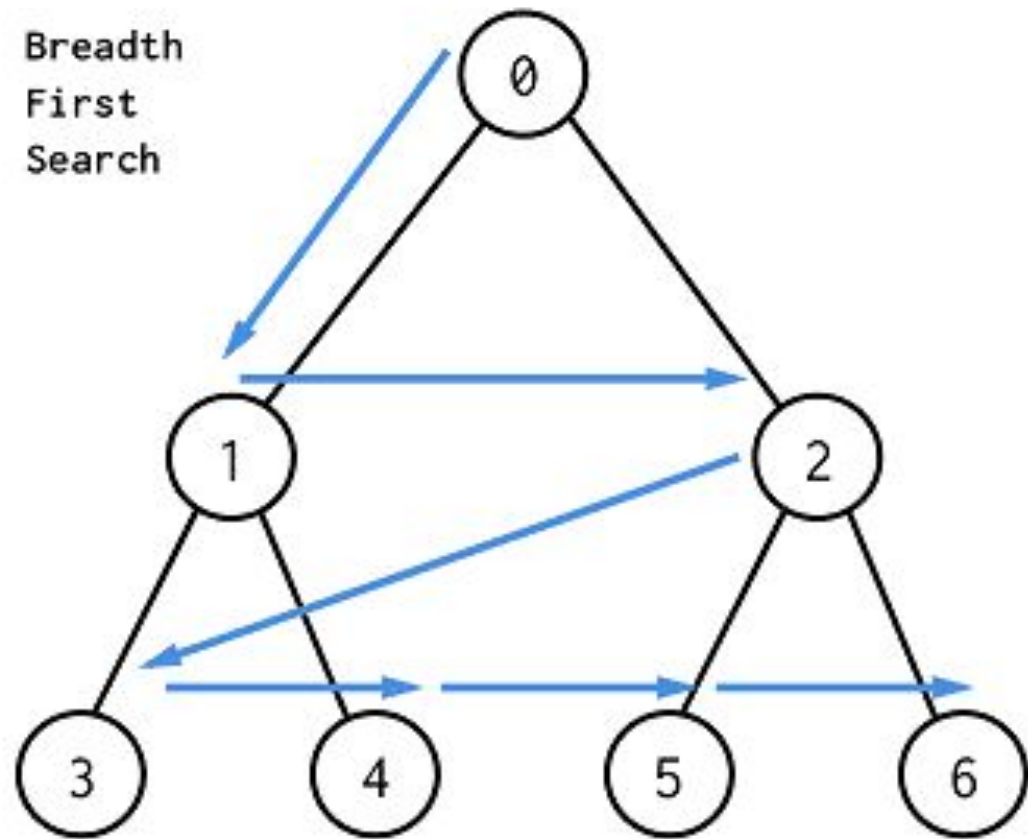
- **Concept:** DFS starts at a root node and explores as deeply as possible along a single path, backtracking and exploring other branches only when it reaches a dead end (no unvisited neighbors). It uses a **stack** data structure to track the exploration path, following a "Last In, First Out" (LIFO) principle.

- **Applications in Design Analysis:**

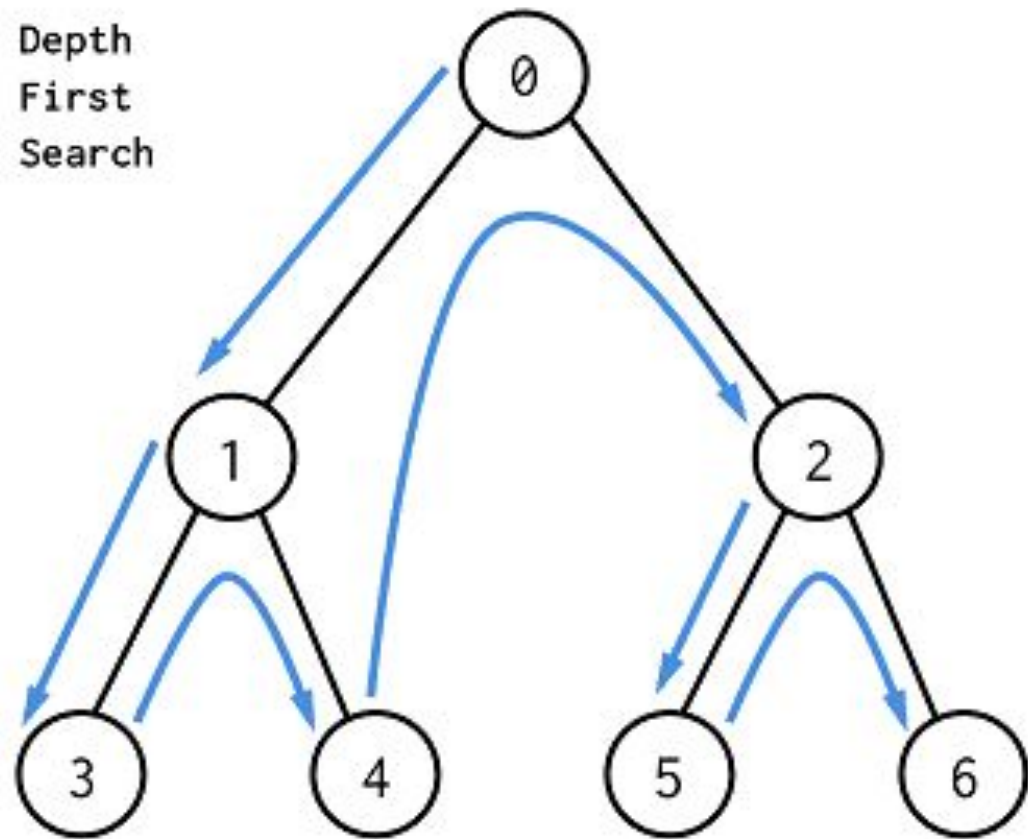
- Finding cycles in a graph: DFS can efficiently detect cycles in a graph by encountering a previously visited node during exploration.
- Topological sorting: In directed acyclic graphs (DAGs), DFS can be used for topological sorting, which orders nodes such that for every directed edge from u to v , u appears before v in the ordering.
- Finding connected components: Similar to BFS, DFS can identify connected components in a graph, but it explores each component in depth rather than level by level.

- **Input:** A graph G and a starting node s .
- **Output:** Visit all nodes in the graph in DFS order.
- **Data Structures:**
 - Stack: To keep track of the exploration path.
 - Visited array: To keep track of visited nodes.
- **Algorithm:**
 - Initialize an empty stack and a visited array, all elements set to False.
 - Push the starting node s onto the stack.
 - Mark s as visited.
 - While the stack is not empty:
 - Pop a node u from the stack.
 - Process node u (e.g., print it).
 - For each unvisited neighbor v of u :
 - Mark v as visited.
 - Push v onto the stack.

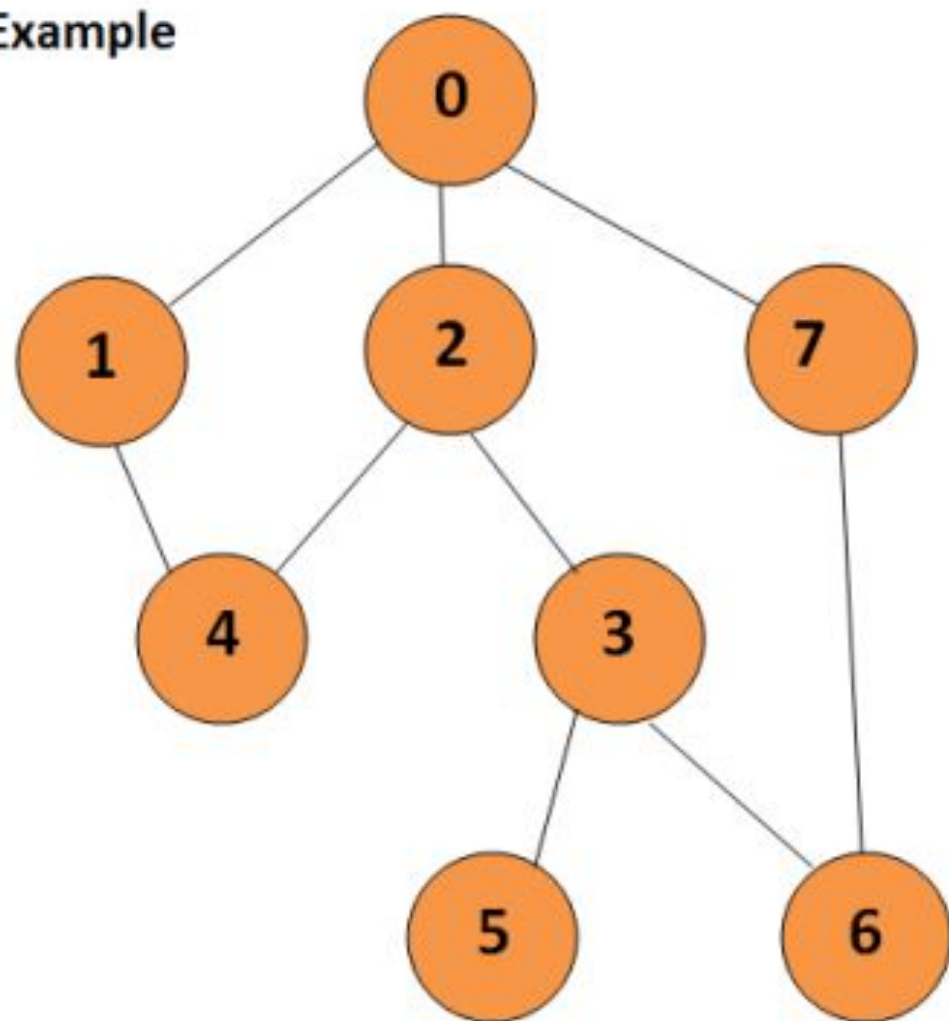
Breadth
First
Search



Depth
First
Search



Graph Example



Edges

0 1

0 2

0 7

1 4

2 3

2 4

3 5

3 6

6 7

DFS:

0 1 4 2 3 5 6 7

BFS:

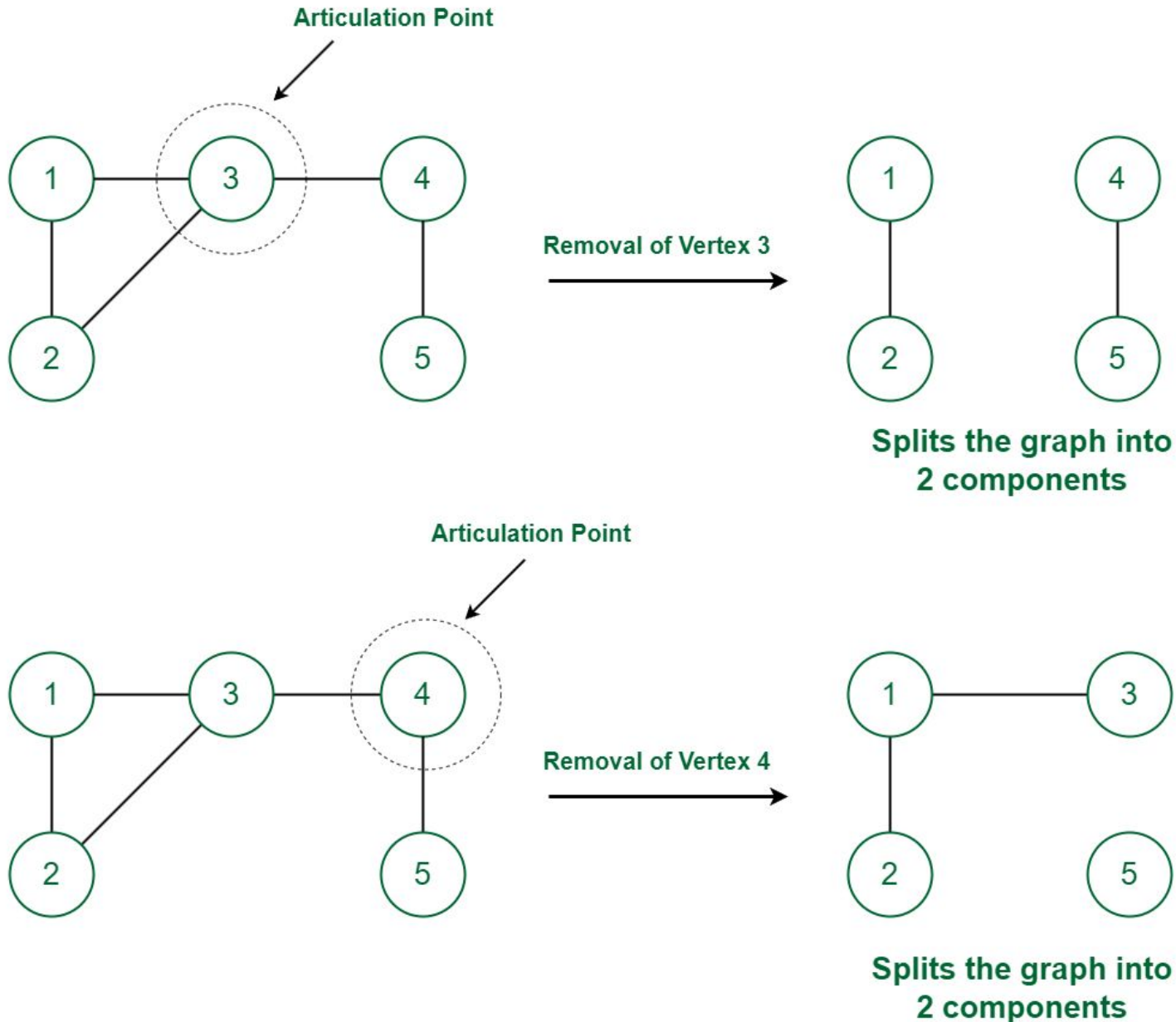
0 1 2 7 4 3 6 5

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest	DFS is used in various applications such as acyclic graphs and finding

Algorithm	Traversal order	Data structure used	Time complexity	Space complexity
DFS	Depth-first	Stack	$O(V+E)$	$O(V)$
BFS	Breadth-first	Queue	$O(V+E)$	$O(V)$

Articulation Points

- If removing a vertex and its associated edges, may cause a disconnect in a graph, then it is called an **articulation point**.



How to identify it?

1. We use DFS(Depth First Search) algorithm to find articulation points.
2. First, select one node to travel through the graph with the DFS algorithm. Then draw the DFS Spanning Tree which is the tree of the traverse.
3. If there are any back edges, draw them in the DFS Spanning Tree. Back Edge means an edge that connects a vertex to the vertex which gets traversed before its parent vertex.
4. Then mark the depth index for each vertex. The depth index means the index that we reach at each vertex when we travel through the DFS Spanning tree.

5. Find out the lowest depth index that we can reach for each vertex.

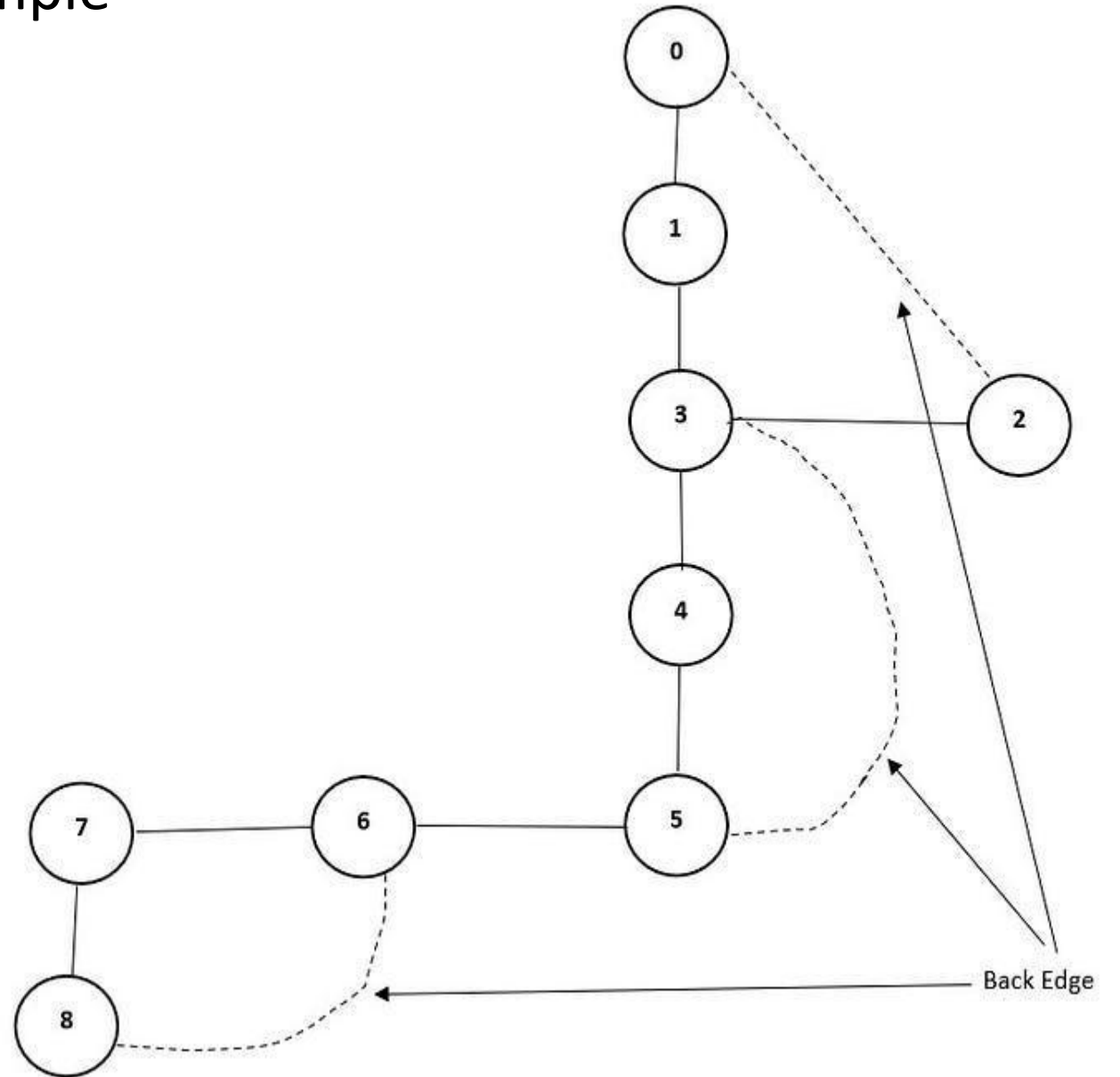
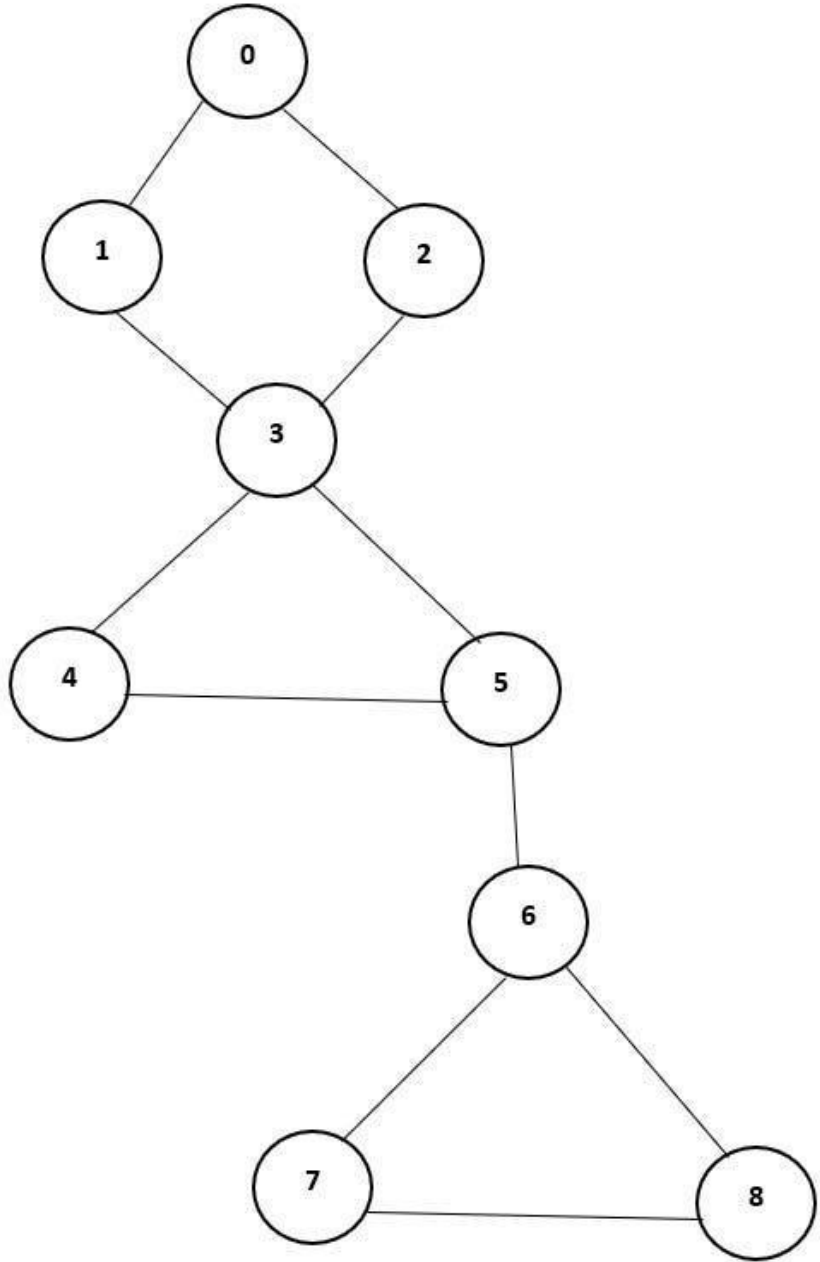
- **rule**

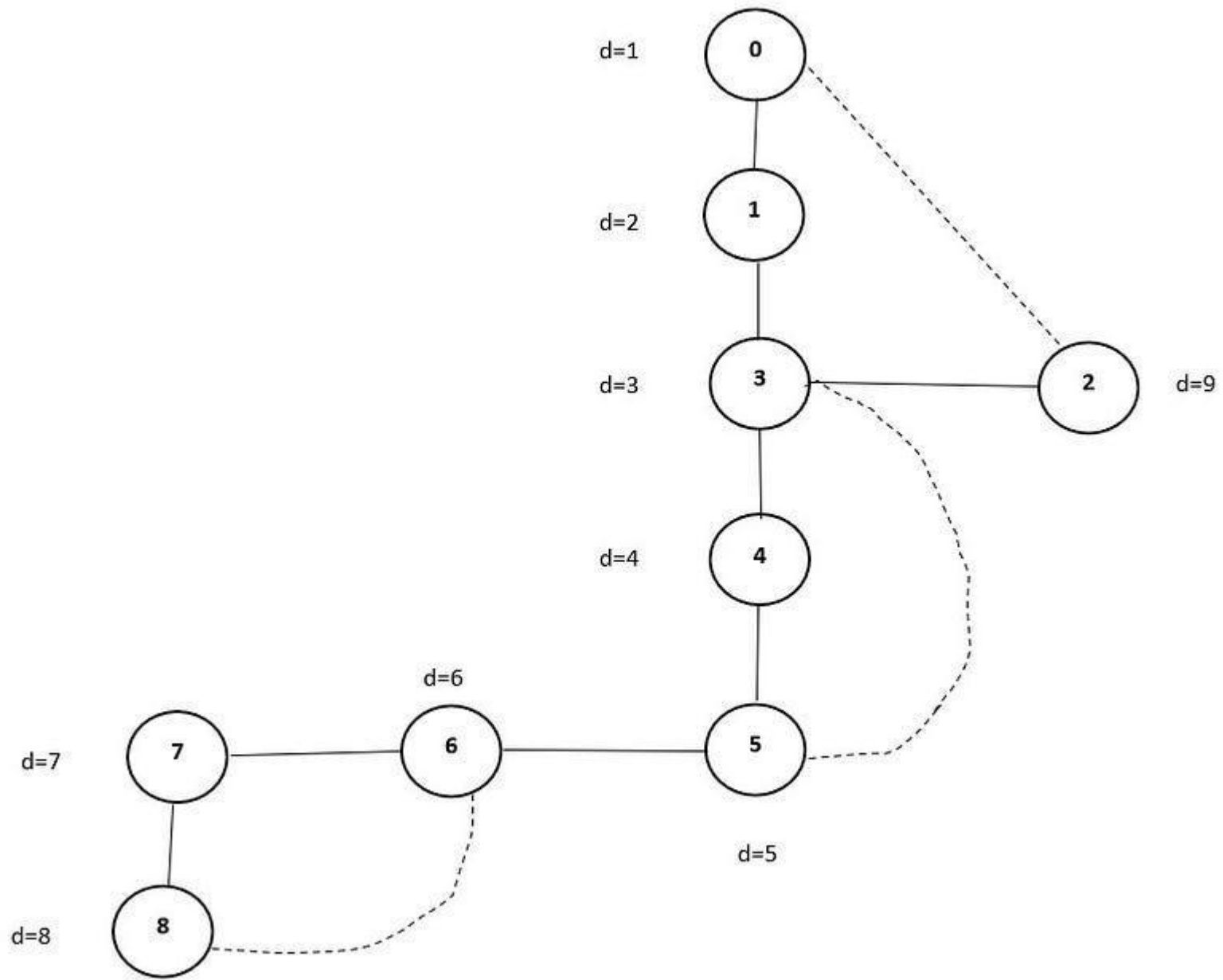
- Only one back edge can be used to reach the lowest depth.

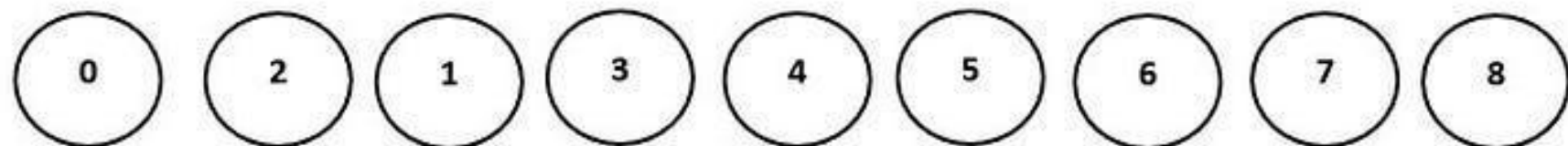
6. Apply the formula to find articulation points.

- If u is a parent node and v is the child node of u :
- **$\text{lowest depth index}[v] \geq \text{depth index}[u]$**
- Then u is an articulation point.
- This formula cannot be applied to a root node.

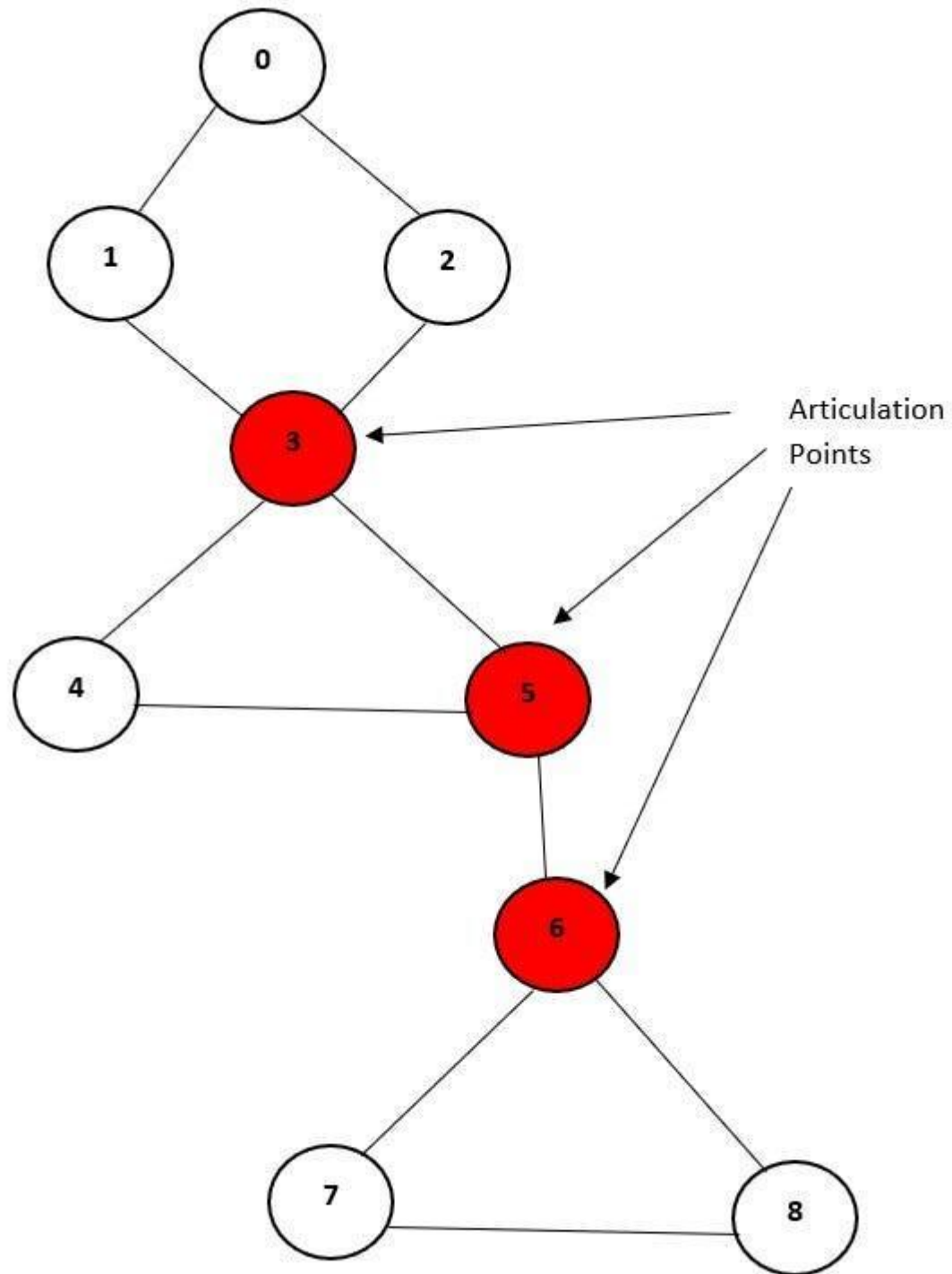
Example







Depth Index	1	9	2	3	4	5	6	7	8
Lowest Index	1	1	1	1	3	3	6	6	6



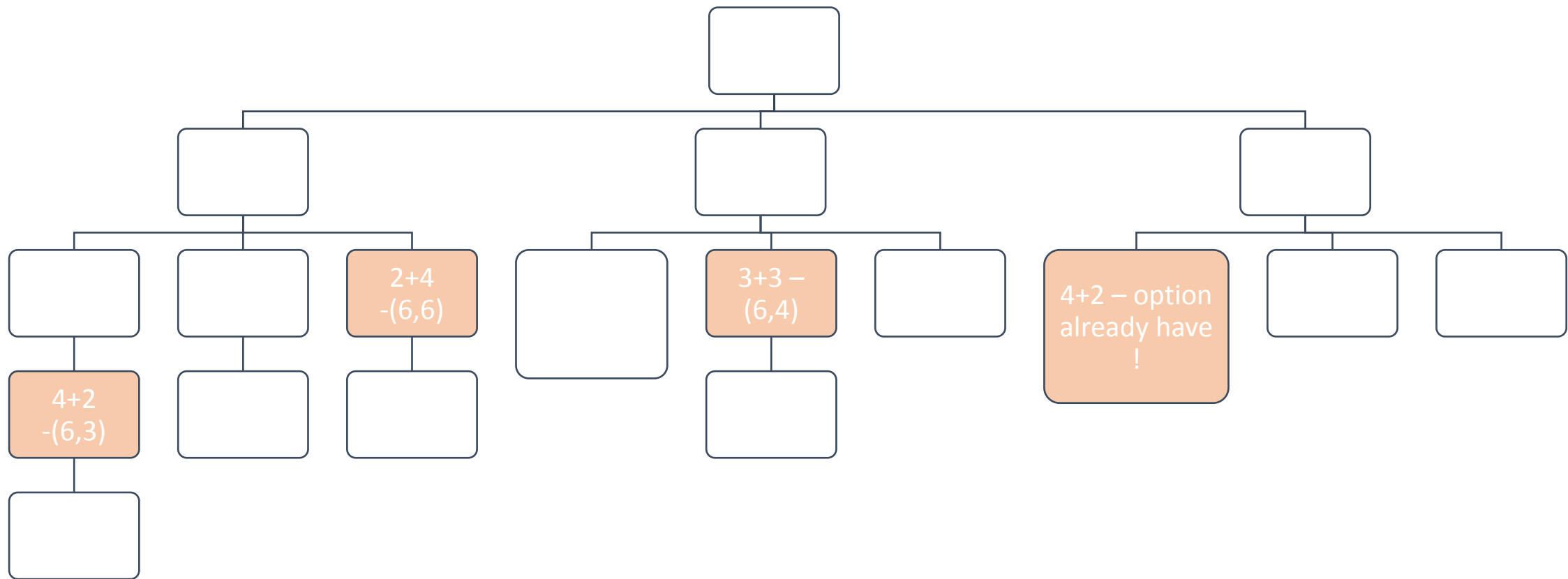
- If u is a parent node and v is the child node of u :
- **lowest depth index $[v] \geq \text{depth index } [u]$**
- Then u is an articulation point

Backtracking

- Backtracking is a technique used in algorithm design to systematically search for solutions to a problem by exploring all possible paths and backtracking from those paths as soon as it determines that the path cannot lead to a valid solution. It is often used in problems involving optimization, constraint satisfaction, and combinatorial search.

Knapsack Problem - Backtracking

- Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items such that the total weight does not exceed a given limit (the capacity of the knapsack).
- This problem can be solved using dynamic programming or backtracking techniques.

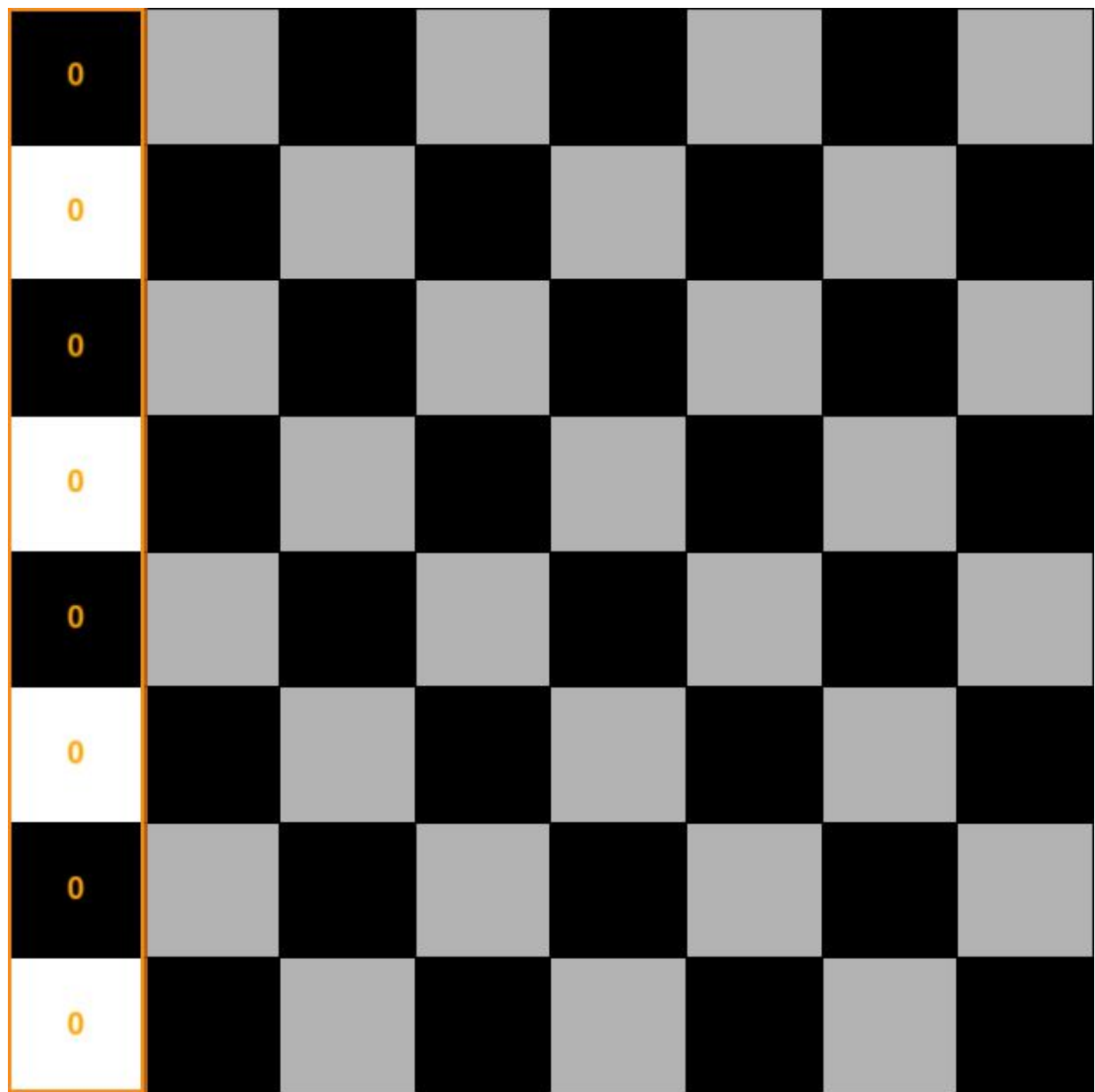





Item No	Item-1	Item-2	Item-3
Weight	2	3	4
Profit or value	1	2	5

Total Weight =6





Eight Queens Problem - Backtracking

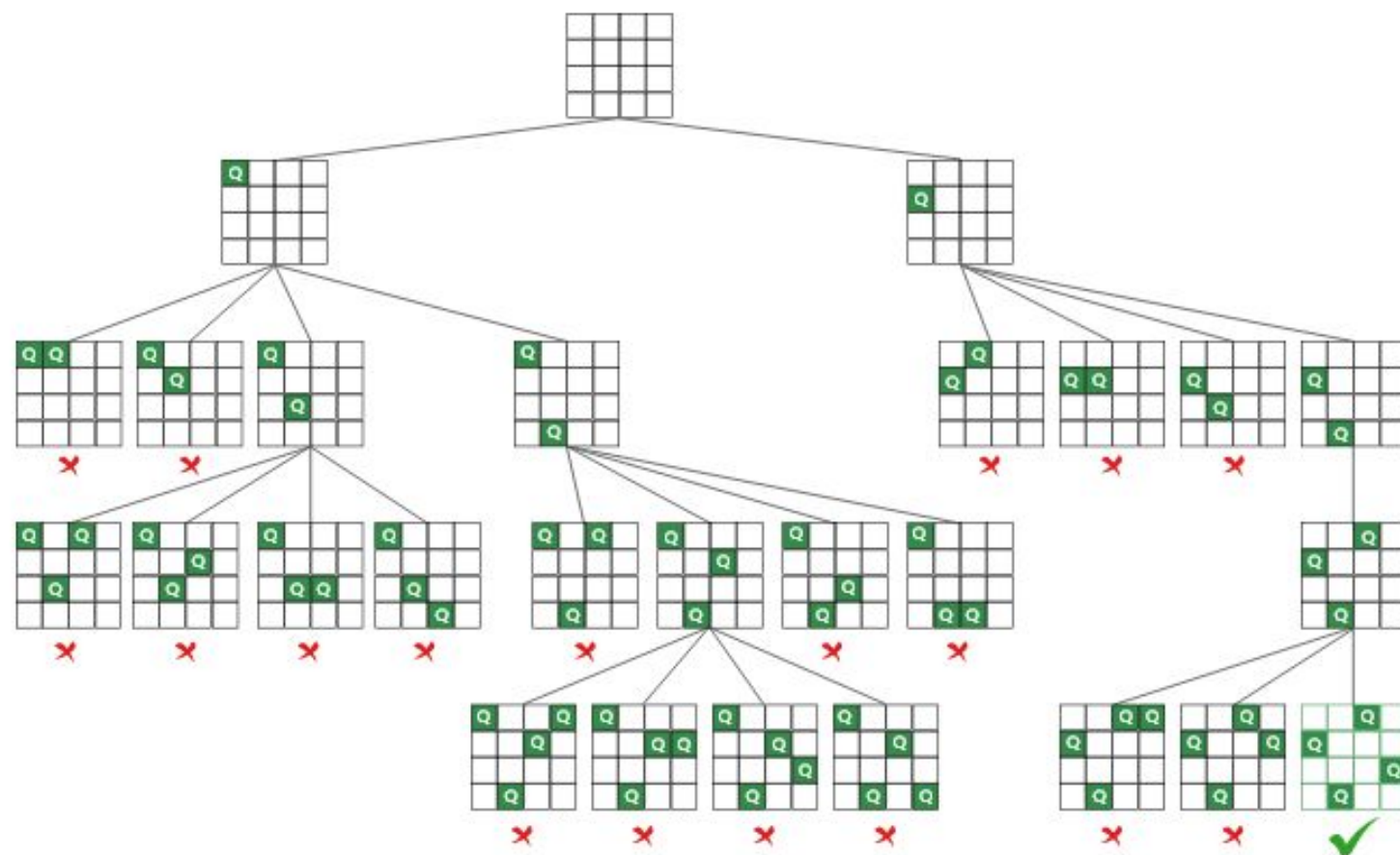
- The Eight Queens Problem is a classic chessboard problem where the task is to place eight queens on a standard 8x8 chessboard in such a way that no two queens threaten each other.
- This problem can be solved using backtracking. The algorithm tries all possible arrangements of queens on the board, and at each step, it checks whether the current arrangement is valid. If not, it backtracks and tries a different arrangement.



Answer of 4 queen problem



8 queen problem

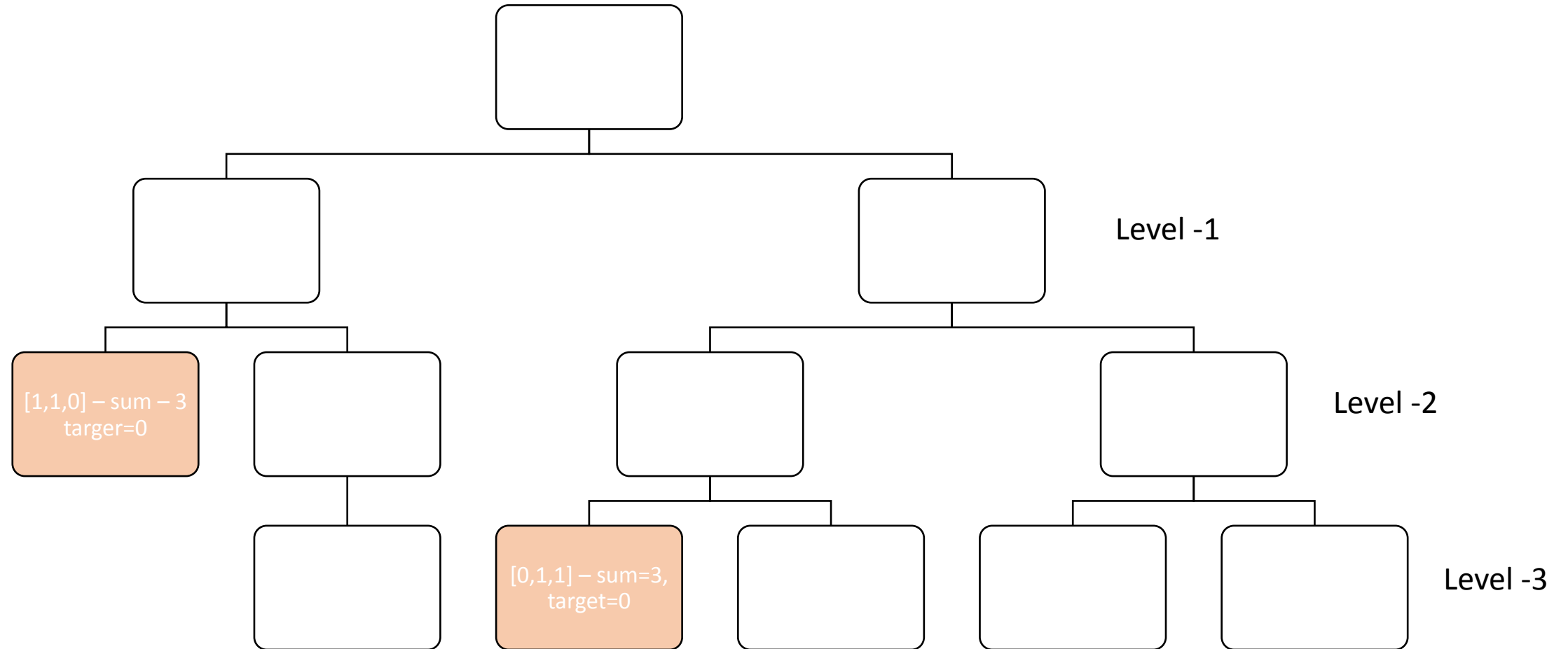
	1	2	3	4	5	6	7	8
1				q ₁				
2						q ₂		
3								q ₃
4		q ₄						
5							q ₅	
6	q ₆							
7			q ₇					
8					q ₈			

Sum of Subset Problem - Backtracking

- Given a set of positive integers and a target sum, determine whether there is a subset of the given set whose sum is equal to the target sum.
- This problem can also be solved using backtracking. The algorithm recursively explores all possible subsets of the given set and checks whether their sum equals the target sum.

Input: set[] = {1,2,1}, sum = 3

Output: [1,2],[2,1]



- **Input:** $set[] = \{5, 10, 12, 13, 15, 18\}$, $sum = 30$
- **Output:** $\{1, 1, 0, 0, 1, 0\}, \{1, 0, 1, 1, 0, 0\}, \{0, 0, 1, 0, 0, 1\}$

Branch and Bound

- Branch and Bound is a problem-solving algorithm that systematically searches through the space of potential solutions to an optimization problem. It is particularly useful for solving combinatorial optimization problems where the goal is to find the best solution from a finite set of possibilities. The algorithm divides the problem space into smaller subspaces (branches) and uses bounding techniques to efficiently eliminate branches that cannot lead to an optimal solution.

Initialization:	The algorithm begins by initializing a priority queue or a data structure to keep track of partial solutions. Each partial solution contains information about the current state of the problem, such as the current solution vector, the cost or value of the solution, and any additional information needed to evaluate the potential of the solution.
Branching:	The algorithm selects one or more variables (or dimensions) of the problem space to branch on. It generates new partial solutions by exploring different choices or assignments for these variables. This branching process creates multiple branches, each representing a different subproblem or a partial solution.
Bounding:	After branching, the algorithm uses bounding techniques to estimate the potential value or cost of each partial solution. These bounds help in determining whether a branch is promising or not. If a branch's estimated value is worse than the current best solution found so far, it can be pruned, i.e., discarded from further consideration. This pruning process helps in reducing the search space and improving the efficiency of the algorithm.
Exploration:	The algorithm continues to explore the search space by recursively branching on promising subproblems. At each step, it selects the most promising branch from the priority queue and repeats the branching and bounding process until all branches have been explored or pruned.
Termination:	The algorithm terminates when all branches have been explored, or when a certain termination criterion is met (e.g., reaching a specified time limit or finding a solution that meets a certain threshold).
Solution Reconstruction:	Once the algorithm terminates, it returns the best solution found during the exploration process.

Types of Branch and Bound Solutions:

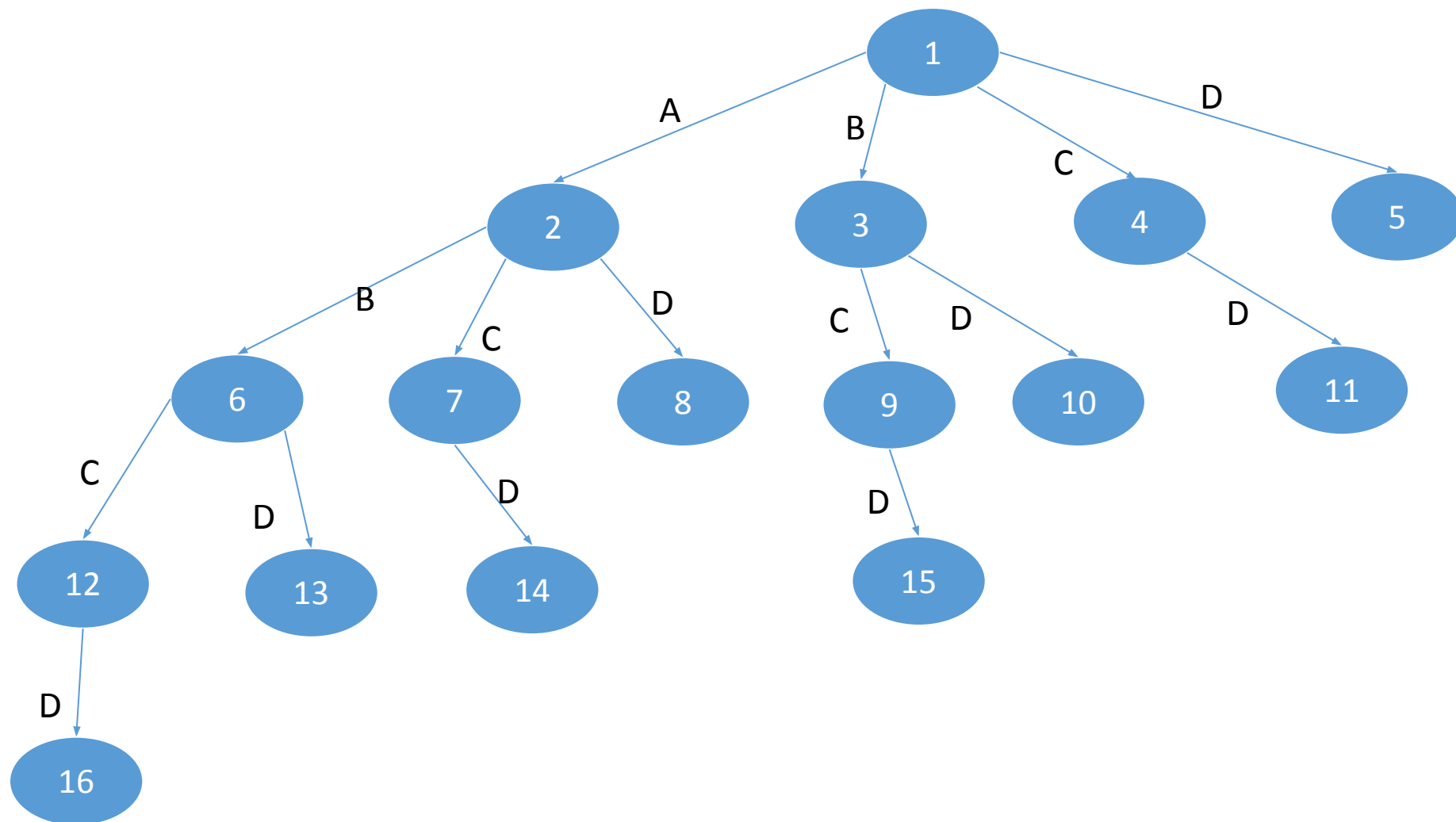
- **Variable size solution:**
 - For example $\{A, B, C, D\}$ - solution is $\{A, B\}$.
- **Fixed-size solution:**
 - For example, $\{A, B, C, D\}$ - solution is $\{1,1,0,0\}$.

Classification of Branch and Bound Problems:

- FIFO Branch and Bound
- LIFO Branch and Bound
- Least Cost-Branch and Bound

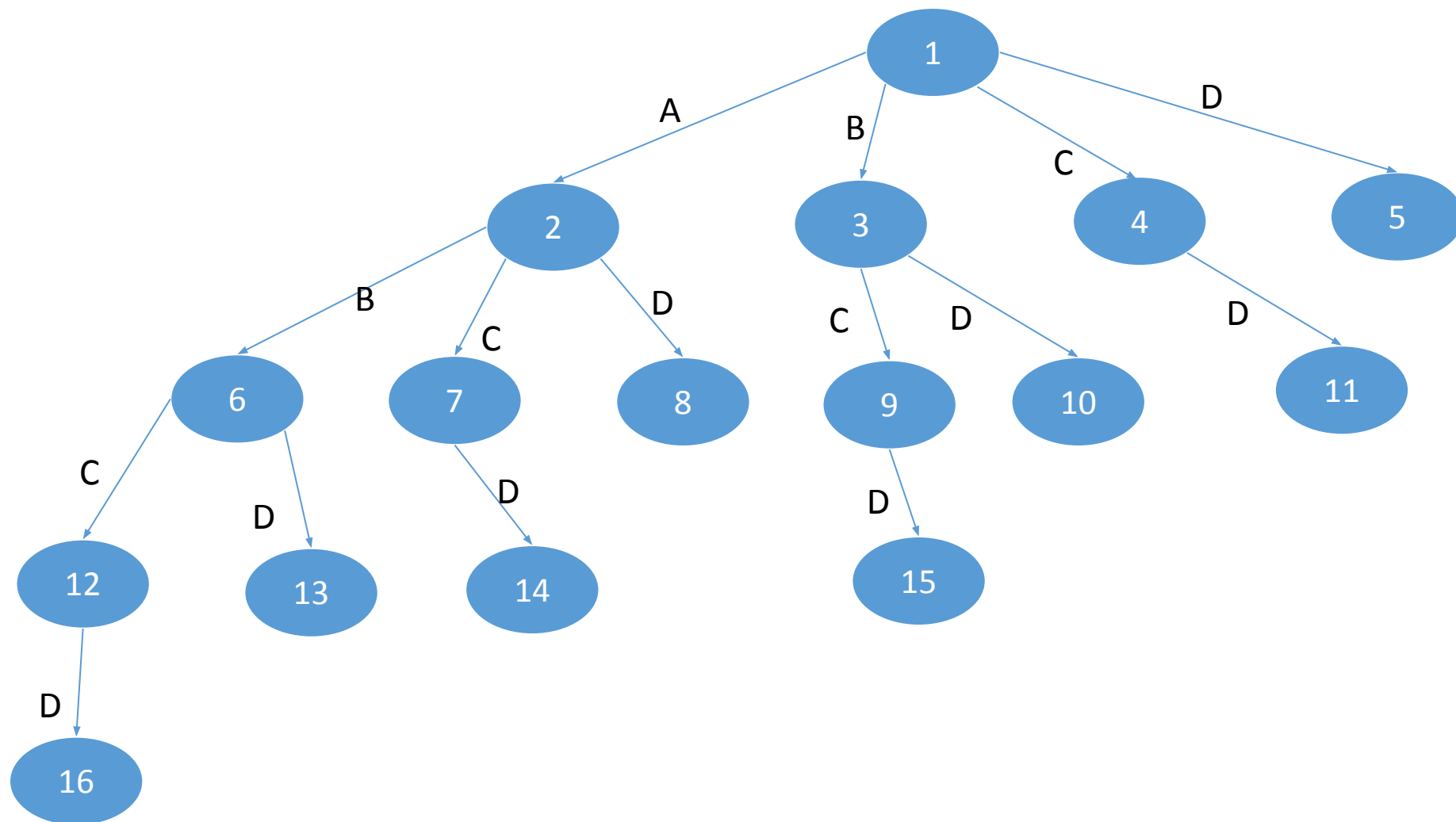
FIFO Branch and Bound

- First-in-first-out is an approach to the branch and bound problem that uses the queue approach to create a state-space tree. In this case, the breadth-first search is performed, that is, the elements at a certain level are all searched, and then the elements at the next level are searched, starting with the first child of the first node at the previous level.
- For a given set {A, B, C, D}, the state space tree will be constructed -



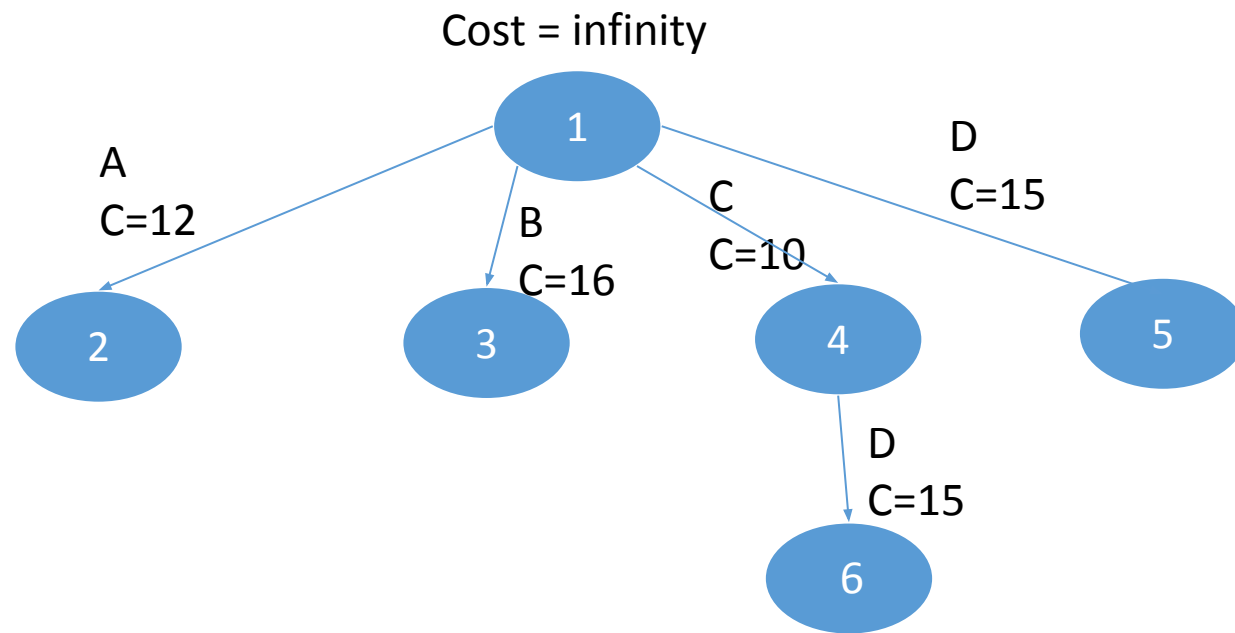
LIFO Branch and Bound

- The Last-In-First-Out approach for this problem uses stack in creating the state space tree. When nodes are added to a state space tree, they are added to a stack. After all nodes of a level have been added, we pop the topmost element from the stack and explore it.
- For a given set {A, B, C, D}, the state space tree will be constructed as follows



Least Cost-Branch and Bound

- To explore the state space tree, this method uses the cost function. The previous two methods also calculate the cost function at each node but the cost is not been used for further exploration.
- In this technique, nodes are explored based on their costs, the cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.
Now, Consider a node whose cost has been determined. If this value is greater than U_0 , this node or its children will not be able to give a solution. As a result, we can kill this node and not explore its further branches. As a result, this method prevents us from exploring cases that are not worth it, which makes it more efficient for us.
- Let's first consider node 1 having cost infinity shown below:
- In the following diagram, node 1 is expanded into four nodes named 2, 3, 4, and 5.



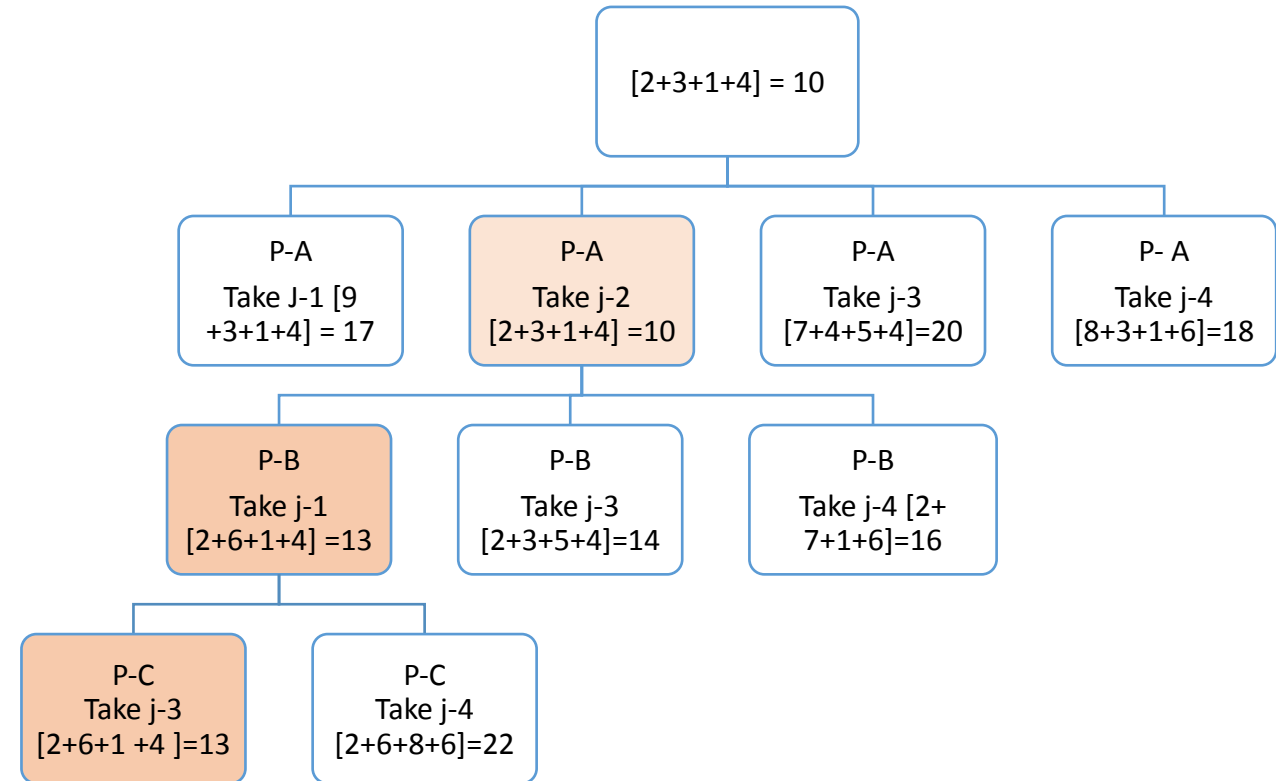
Assignment problem

- **Problem Statement:** Given a set of workers and tasks, each with a cost or weight associated with it, the assignment problem aims to find the most efficient way to assign workers to tasks, such that each task is assigned to exactly one worker and each worker is assigned to exactly one task, while minimizing the total cost or maximizing the total profit.

Minimum Cost = $2 + 3 + 1 + 4 = 10$

Assignment problem

	J-1	J-2	J-3	J-4
P-A	9	2	7	8
P-B	6	4	3	7
P-C	5	8	1	8
P-D	7	6	9	4



Final selected jobs

	J-1	J-2	J-3	J-4
P-A	9	2	7	8
P-B	6	4	3	7
P-C	5	8	1	8
P-D	7	6	9	4

Suppose we have to solve the instance whose cost matrix is shown in Figure 9.13. To obtain an upper bound on the answer, note that $a \rightarrow 1, b \rightarrow 2, c \rightarrow 3, d \rightarrow 4$ is one possible solution whose cost is $11 + 15 + 19 + 28 = 73$. The optimal solution to the problem cannot cost more than this. Another possible solution is $a \rightarrow 4, b \rightarrow 3, c \rightarrow 2, d \rightarrow 1$ whose cost is obtained by adding the elements in the other diagonal of the cost matrix, giving $40 + 13 + 17 + 17 = 87$. In this case the second solution is no improvement over the first. To obtain a lower bound on the solution, we can argue that whoever executes task 1, the cost will be at least 11; whoever executes task 2, the cost will be at least 12, and so on. Thus adding the smallest elements in each column gives us a lower bound on the answer. In the example, this is $11 + 12 + 13 + 22 = 58$. A second lower bound is obtained by adding the smallest elements in each row, on the grounds that each agent must do something. In this case we find $11 + 13 + 11 + 14 = 49$, not as useful as the previous lower bound. Pulling these facts together, we know that the answer to our instance lies somewhere in $[58..73]$.

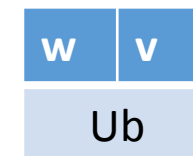
	1	2	3	4
<i>a</i>	11	12	18	40
<i>b</i>	14	15	13	22
<i>c</i>	11	17	19	23
<i>d</i>	17	14	20	28

Figure 9.13. The cost matrix for an assignment problem

Knapsack Problem

- Fill the knapsack of capacity W , with a given set of items I_1, I_2, I_3 having weights w_1, w_2, w_3 in such a manner that the total weight of the items should not exceed the knapsack capacity and maximum value can be obtained.
- Items are arranged in descending order of value per weight ratio.
- We have bound that none of these items can have total sum more than knapsack capacity.
- The tree is constructed as binary tree, where left branch signifies the inclusion of the item and right branch signifies exclusion.

Structure of the node is: $U_b = v + (W - w) * (v_{i+1} / w_{i+1})$



Items	w	v	vi/wi
I1	1	2	2
I2	2	3	1.5
I3	3	4	1.3

W=3

1. Start from the root node.
upper bound is :

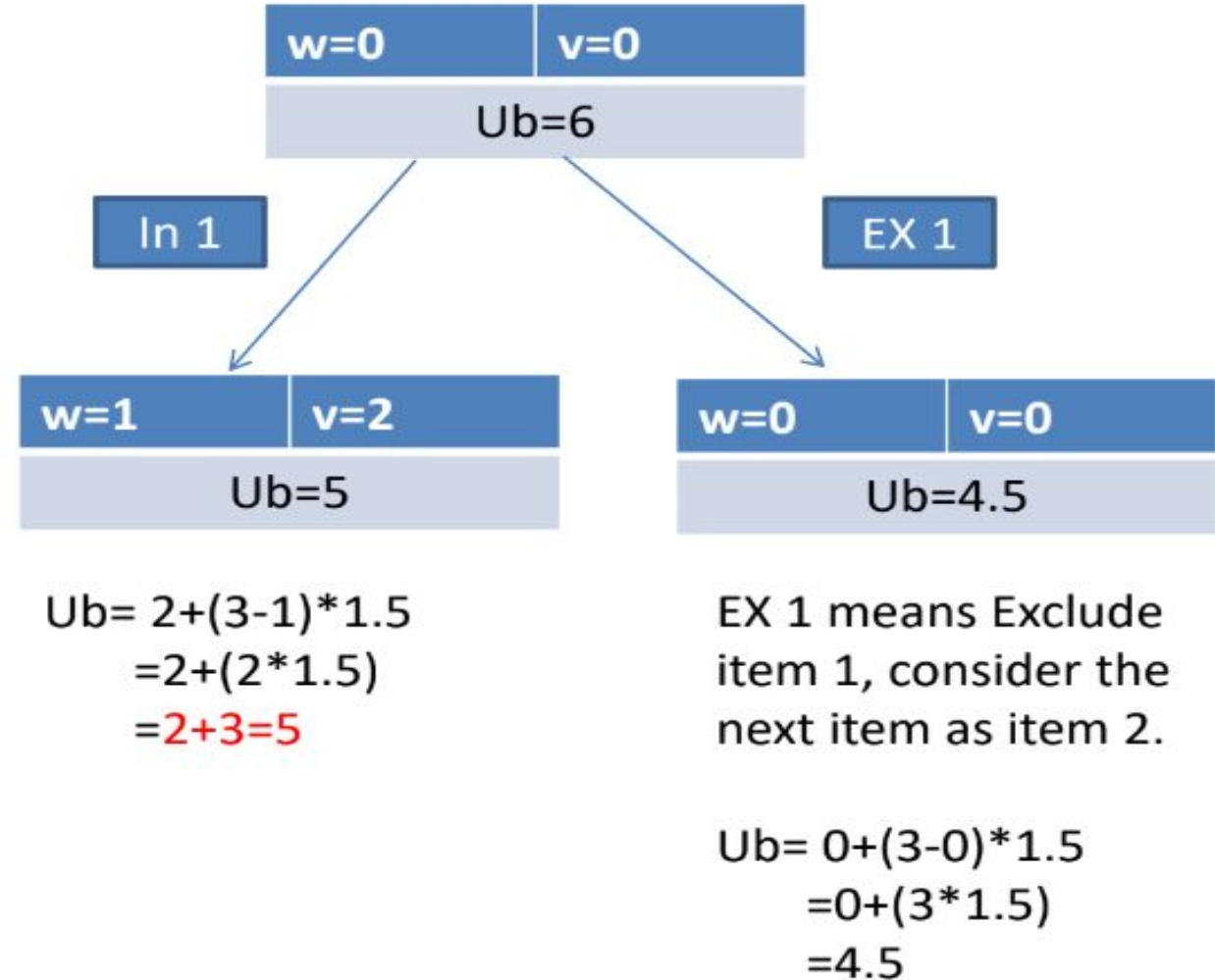
$v=0, w=0, W=3, v_1/w_1=2$

$$Ub = 0 + (3-0) * 2$$

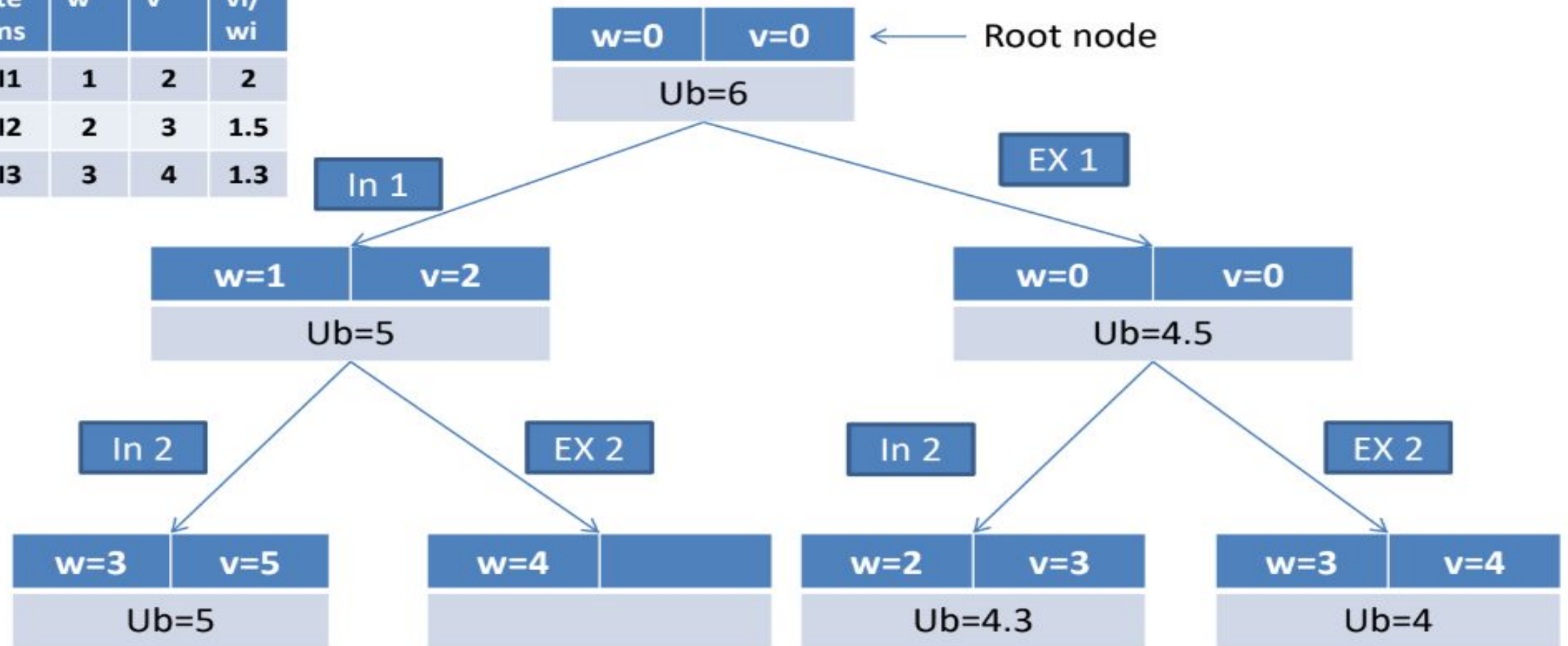
node is now:

w=0	v=0
Ub=6	

2. Include item 1 indicated by the left branch, and exclude item 1, which is indicated by right branch.



Items	w	v	v_i/w_i
I1	1	2	2
I2	2	3	1.5
I3	3	4	1.3



- At every level, compute the upper bound, and explore the node while selecting the item.
- Finally, the node with maximum upper bound is selected as an optimal solution.
- In this example, item 1 and item 2 gives optimum solution.

	Branch and Bound	Backtracking
Objective	To find the optimal solution to an optimization problem. It systematically explores the entire search space, pruning unpromising branches based on bounds, until it finds the best possible solution.	To find any feasible solution to a problem. It explores the search space recursively, backtracking from dead ends (i.e., partial solutions that cannot be extended into a complete solution), until it finds a solution or exhausts all possibilities.
Pruning Mechanism	Branch and Bound uses bounding techniques to prune unpromising branches from the search space. It computes lower and upper bounds for each partial solution and discards branches whose bounds indicate that they cannot lead to an optimal solution.	Backtracking does not use bounding techniques like Branch and Bound. Instead, it relies on early detection of dead ends while exploring the search space. When a dead end is encountered (e.g., a partial solution violates a constraint or cannot be extended further), backtracking occurs, and the algorithm retreats to the previous decision point to explore other possibilities.

	Branch and Bound	Backtracking
Optimality	Branch and Bound guarantees finding the optimal solution to an optimization problem, provided that the bounding techniques are correctly implemented and the search space is explored completely.	Backtracking does not guarantee finding the optimal solution. It may find a feasible solution quickly, but it can also continue searching indefinitely if the search space is large or the problem structure is complex.
Usage	Branch and Bound is typically used for solving optimization problems such as the Traveling Salesman Problem, the Knapsack Problem, and the Assignment Problem, where finding the best solution is crucial.	Backtracking is commonly used for constraint satisfaction problems (CSPs) such as the N-Queens Problem, Sudoku, and graph coloring, where the goal is to find any feasible solution rather than the best one.

String Matching

■ Application

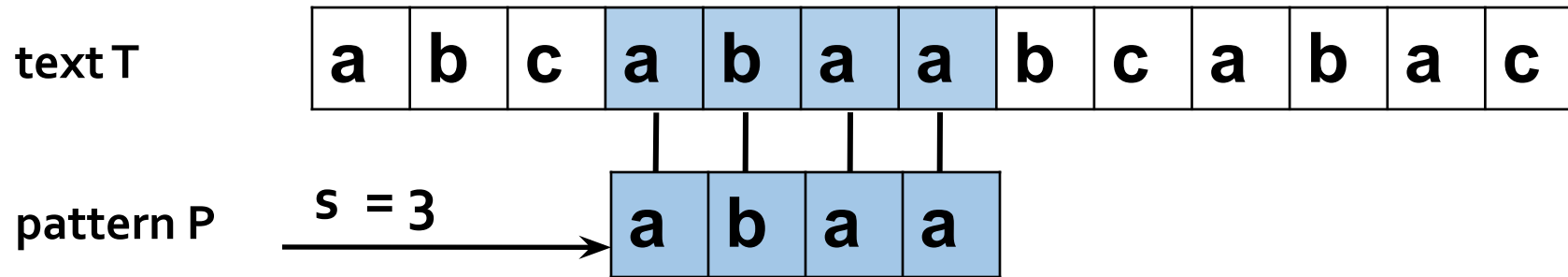
- 1) Text Editing programs.

- 2) To search particular patterns in DNA sequences.

Assumptions

- For the String matching problem.....
- Text is an array $T[1..n]$ of length n .
- Pattern is an array $P[1..m]$ of length m .
- $m \leq n$
- Elements of P and T are characters from a finite alphabet Σ .
- The array P and T are called strings of characters where $\Sigma = \{0,1\}$ or $\Sigma = \{a,b,\dots,z\}$

Valid shift



Pattern p occurs with shift s in text T if $0 \leq s \leq n - m$ and $T[s+1..s+m] = P[1..m]$

If P occurs with shift s in T , we can call s a valid shift, otherwise invalid shift.

What is the String matching Problem??????

The string matching problem is the problem of finding all valid shifts which a given pattern P occurs in given text T .

String matching Algorithm

- 1) Naïve brute force algorithm.
- 2) Robin and karp algorithm.
- 3) String matching algorithm using finite autometa.
- 4) KMP algorithm.

Except first algorithm, remaining algorithm perform some preprocessing based on the pattern and then find all valid shift.

Notation and Terminology

- Σ^* \rightarrow set of all finite length strings formed using characters from alphabet Σ .
- $\varepsilon \rightarrow$ zero length empty string ,belongs to Σ^*
- $|x| \rightarrow$ length of string x .
- $xy \rightarrow$ concatenation of two string x and y .
- $|x| + |y| \rightarrow$ length of $x + y$ and consist characters from x followed by the characters from y .

Notation and Terminology (cont.....)

- A string w is a prefix of a string x , denoted $w \sqsubset x$, if $x=wy$ for some string $y \in \Sigma^*$.

If $w \sqsubset x$, then $|w| \leq |x|$.

- A string w is a suffix of a string x , denoted $w \sqsupset x$, if $x=yw$ for some string $y \in \Sigma^*$.

If $w \sqsupset x$, then $|w| \leq |x|$.

- The empty string ε is both a suffix and prefix of every string.

Notation and Terminology (cont.....)

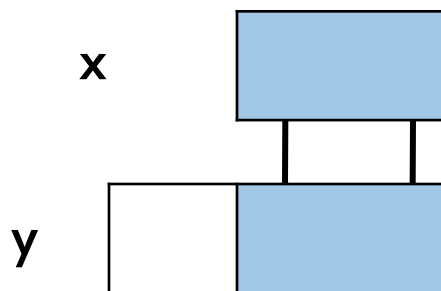
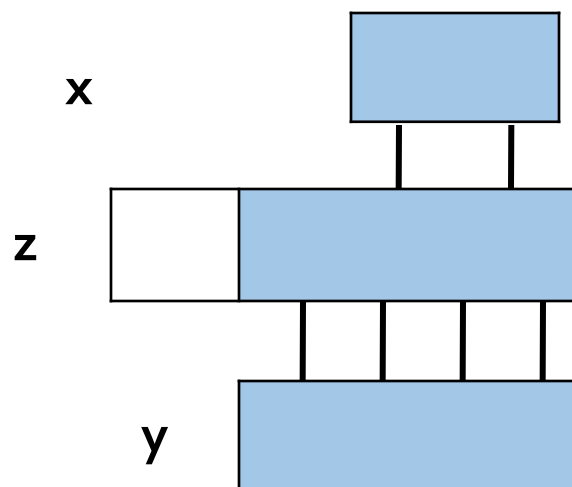
- $ab \sqsubseteq abcca$
- $cca \sqsupseteq abcca$
- We have $x \sqsupseteq y$ if and only if $xa \sqsupseteq ya$.
- \sqsubseteq and \sqsupseteq are transitive relations.
- Lemma :-

suppose that x, y, z are strings such that $x \sqsupseteq z$
and $y \sqsupseteq z$.

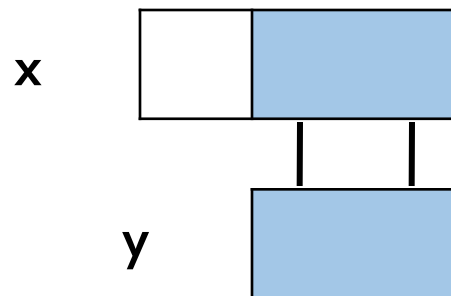
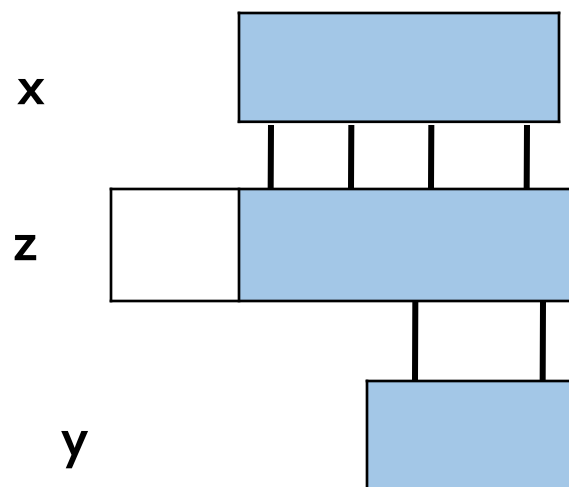
if $|x| \leq |y|$, then $x \sqsupseteq y$.

if $|x| = |y|$, then $x = y$.

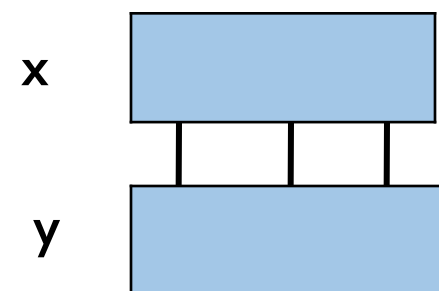
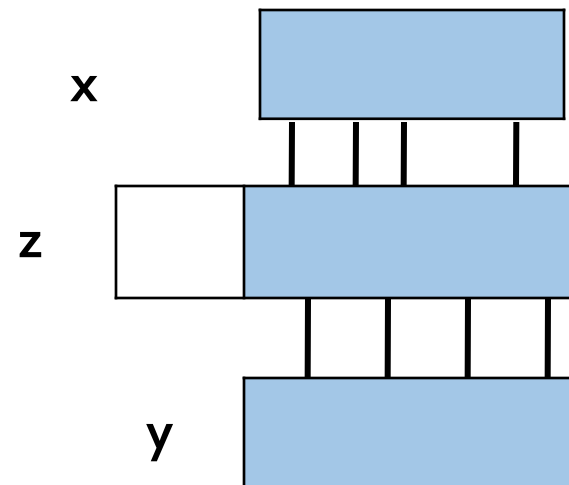
Graphical proof of lemma



(a)



(b)



(c)

The naïve string – matching algorithm

- The naïve algorithm finds all valid shifts.
- Check the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s .

NAÏVE – STRING – MATCHER(T,P)

- 1. $n \leftarrow \text{length}[T]$**
- 2. $m \leftarrow \text{length}[P]$**
- 3. for $s \leftarrow 0$ to $n - m$**
- 4. do if $P[1..m] = T[s+1..s+m]$**
- 5. then print “ Pattern occurs with shift “ s**

The Rabin-Karp Algorithm

- Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.
- Pattern itself
- All the substrings of the text of length m which is the size of pattern.

How is Hash Value calculated in Rabin-Karp?

- **Hash value** is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.

- Here's how the hash value is typically calculated in Rabin-Karp:
- **Step 1:** Choose a suitable **base** and a **modulus**:
 - Select a prime number '**p**' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
 - Choose a base '**b**' (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).
- **Step 2:** Initialize the hash value:
 - Set an initial hash value '**hash**' to **0**.

- **Step 3:** Calculate the initial hash value for the **pattern**:
 - Iterate over each character in the **pattern** from **left** to **right**.
 - For each character '**c**' at position '**i**', calculate its contribution to the hash value as '**c** * (**b**^{**pattern_length - i - 1**}) % **p**' and add it to '**hash**'.
 - This gives you the hash value for the entire **pattern**.
- **Step 4:** Slide the pattern over the **text**:
 - Start by calculating the hash value for the first substring of the **text** that is the same length as the **pattern**.

- **Step 5:** Update the hash value for each subsequent substring:
 - To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
 - The formula for updating the hash value when moving from position '**i**' to '**i+1**' is:
- $$\text{hash} = (\text{hash} - (\text{text}[\text{i} - \text{pattern_length}] * (\text{b}^{\text{pattern_length} - 1})) \% \text{p}) * \text{b} + \text{text}[\text{i}]$$
- **Step 6:** Compare hash values:
 - When the hash value of a substring in the **text** matches the hash value of the **pattern**, it's a **potential match**.
 - If the hash values match, we should perform a character-by-character comparison to confirm the match, as [hash collisions](#) can occur.

- Given Text = 315265 and Pattern = 26
- We choose $b = 11$
- $P \bmod b = 26 \bmod 11 = 4$

3 1 5 2 6 5 $31 \bmod 11 = 9$ not equal to 4

3 1 5 2 6 5 $15 \bmod 11 = 4$ equal to 4 -> spurious hit

3 1 5 2 6 5 $52 \bmod 11 = 8$ not equal to 4

3 1 5 2 6 5 $26 \bmod 11 = 4$ equal to 4 -> an exact match!!

3 1 5 2 6 5 $65 \bmod 11 = 10$ not equal to 4

As we can see, when a match is found, further testing is done to ensure that a match has indeed been found.

- **Limitations of Rabin-Karp Algorithm**

- **Spurious Hit:** When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a **spurious hit**. Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use good [hash function](#). It greatly reduces the spurious hit.

RABIN-KARP-MATCHER(T,P,d,q)

```
1  n ← length[T]
2  m ← length[P]
3  h ←  $d^{m-1} \bmod q$ 
4  p ← 0
5  t0 ← 0
6  for i = 1 to m      //Preprocessing
7  .  do p = ( dp + P[i]) mod q
8  .  t0 = (dt0 + T[i] ) mod q
```

RABIN-KARP-MATCHER(T,P,d,q)

```
9   for s = 0 to n-m      //Matching
10  .   do if p = ts
11  .       then if P[1..m] = T[s+1..s+m]
12  .       then print "Pattern occurs with shift " s
13  .   if s < n-m
14  .       then ts+1 = (d(ts - T[s + 1]h )+T[s + m +1]) mod q
```


Analysis

- Preprocessing time $\square \Theta(m)$
- Matching time $\square \Theta((n-m+1) m)$

KMP (Knuth Morris Pratt)

- We have discussed the Naive pattern-searching algorithm in the [previous post](#). The worst case complexity of the Naive algorithm is $O(m(n-m+1))$. The time complexity of the KMP algorithm is $O(n+m)$ in the worst case.
- *txt* = "AAAAABAAABA"
pat = "AAAA"
*We compare first window of **txt** with **pat***

- *txt = "AAAAABAAABA"*
pat = "AAAA" [Initial position]
We find a match. This is same as [Naive String Matching](#).
- *In the next step, we compare next window of **txt** with **pat**.*
- *txt = "AAAAABAAABA"*
pat = "AAAA" [Pattern shifted one position]
- ***Need of Preprocessing?***

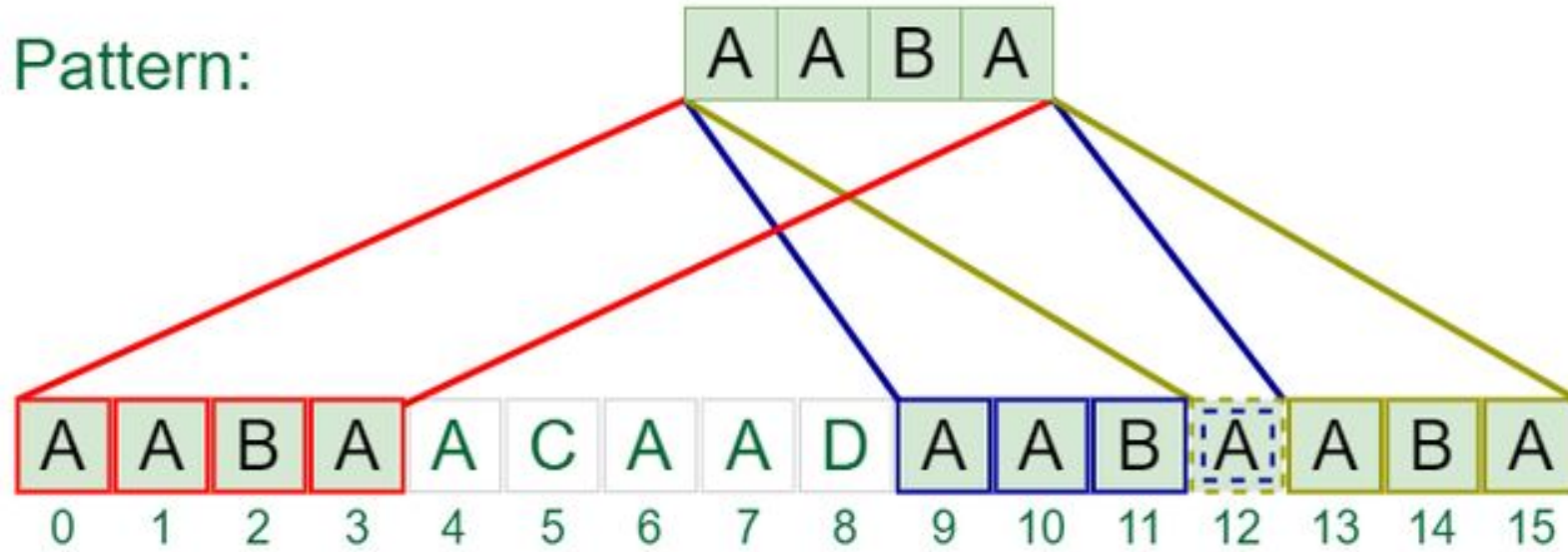
- KMP algorithm preprocesses `pat[]` and constructs an auxiliary **lps[]** of size **m** (same as the size of the pattern) which is used to skip characters while matching.
- Name **lps** indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC", and "ABC".
- We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.
- For each sub-pattern `pat[0..i]` where $i = 0$ to $m-1$, `lps[i]` stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

- For the pattern “AAAA”, $lps[]$ is $[0, 1, 2, 3]$
- For the pattern “ABCDE”, $lps[]$ is $[0, 0, 0, 0, 0]$
- For the pattern “AAACAAAAC”, $lps[]$ is $[0, 1, 2, 0, 1, 2, 3, 3, 3, 4]$

Text: A A B A A C A A D A A B A A B A

Pattern:



Pattern found at index 0, 9, 12

Finite Automata String Matching Algorithm

5-Tuples of Finite Automata

- A *finite automaton* M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
 - Q is a finite set of *states*
 - $q_0 \in Q$ is the *start state*
 - $A \subseteq Q$ is a set of *accepting states*
 - Σ is a finite *input alphabet*
 - δ is the *transition function* that gives the next state for a given current state and input.

Example

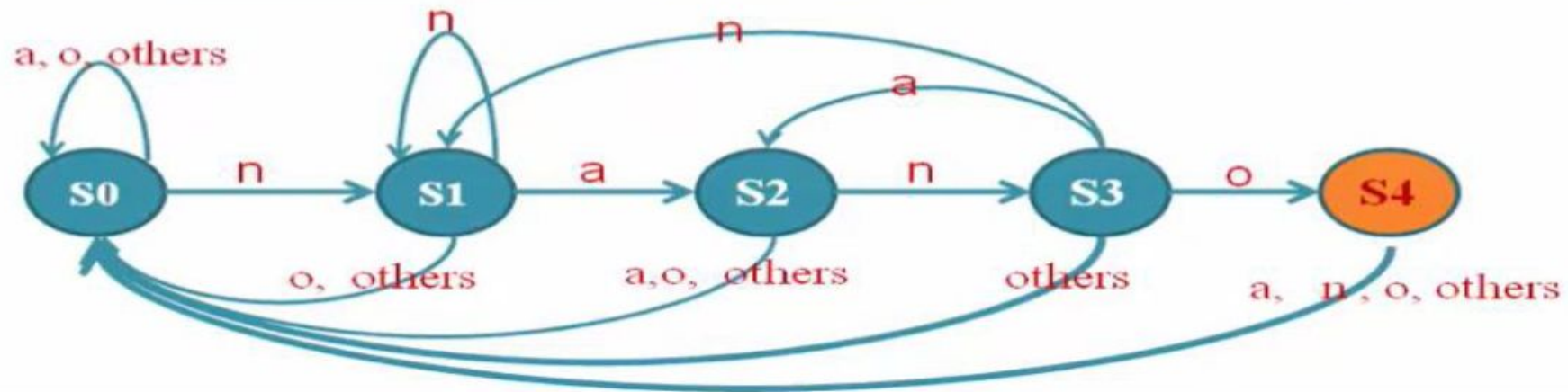
Text = **b a n a n a n o n a**

Pattern = **n a n o**

Transition Table

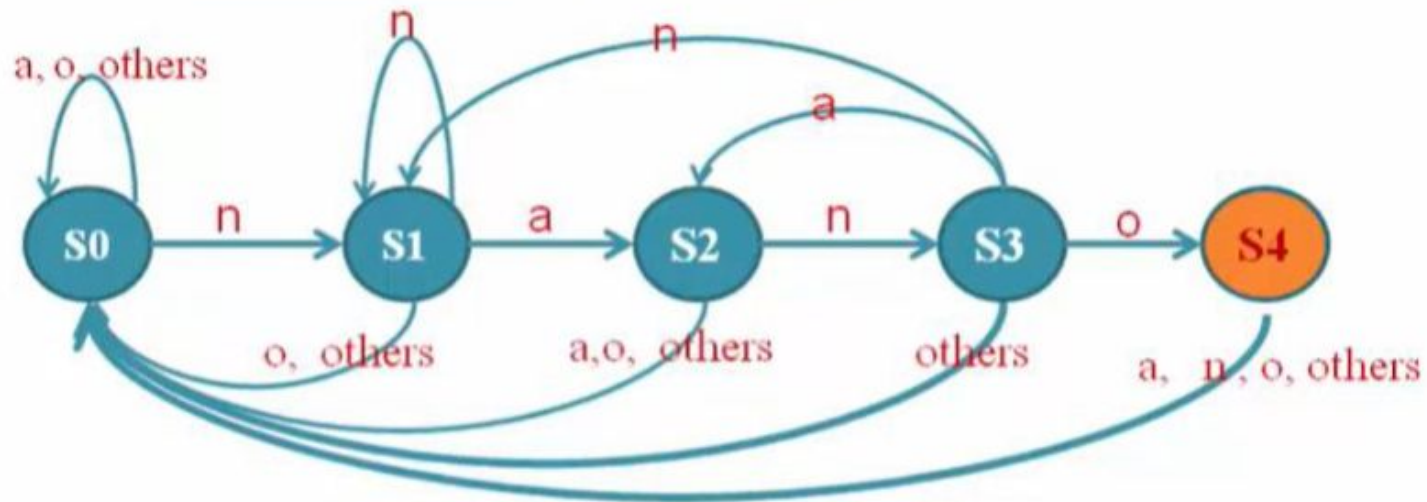
States	n	a	o	others
S0	S1	S0	S0	S0
S1	S1	S2	S0	S0
S2	S3	S0	S0	S0
S3	S1	S2	S4	S0
S4	S0	S0	S0	S0

Finite Automata



Example

Finite Automata



	1	2	3	4	5	6	7	8	9	10
Text	l	a	n	a	n	a	n	o	n	a
S0	S0	S0	S1	S2	S3	S2	S3	S4		

Matching is found

No of shift = $8 - 4 = 4$

How to make Transition Table and Finite Automata of any pattern

- Let's have same example

Text = **b a n a n a n o n a**

Pattern = n a n o

- First Prepare transition table of Pattern :

Identify unique characters from Pattern : n a o

States	n	a	o	others
S0				
S1				
S2				
S3				
S4				



How to make Transition Table and Finite Automata of any pattern

- Let's have same example

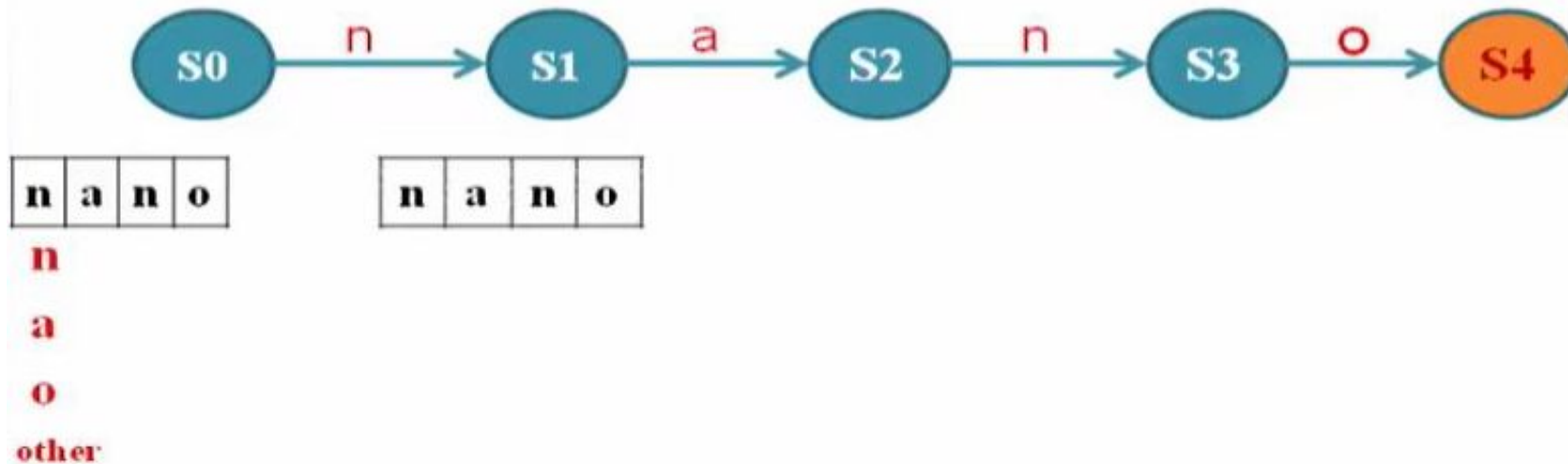
Text = **b a n a n a n o n a**

Pattern = **n a n o**

- First Prepare transition table of Pattern :

Identify unique characters from Pattern : n a o

States	n	a	o	others
S0	S1	S0	S0	S0
S1				
S2				
S3				
S4				



How to make Transition Table and Finite Automata of any pattern

- Let's have same example

Text = **b a n a n a n o n a**

Pattern = **n a n o**

- First Prepare transition table of Pattern :

Identify unique characters from Pattern : n a o

States	n	a	o	others
S0	S1	S0	S0	S0
S1	S1	S2	S0	S0
S2	S3			
S3				
S4				



n a n o

n

a

o

other

n a n o

n n

n n

n a

n a

n o

n o

n a n o

n a n

n a n

n a n

n a a

How to make Transition Table and Finite Automata of any pattern

- Let's have same example

Text = **b a n a n a n o n a**

Pattern = **n a n o**

- First Prepare transition table of Pattern :

Identify unique characters from Pattern : n a o

States	n	a	o	others
S0	S1	S0	S0	S0
S1	S1	S2	S0	S0
S2	S3	S0	S0	S0
S3	S1			
S4				



n a n o

n
a
o
other

n a n o

n n
n n
n a
n a
n o
n o

n a n o

n a n
n a n
n a n
n a a
n a a
n a a

n a n o

n a n n
n a n n
n a n n
n a n n
n a n a
n a n a

Example Text = **b a n a n a n o n a**
 Pattern = **n a n o**

Transition Table

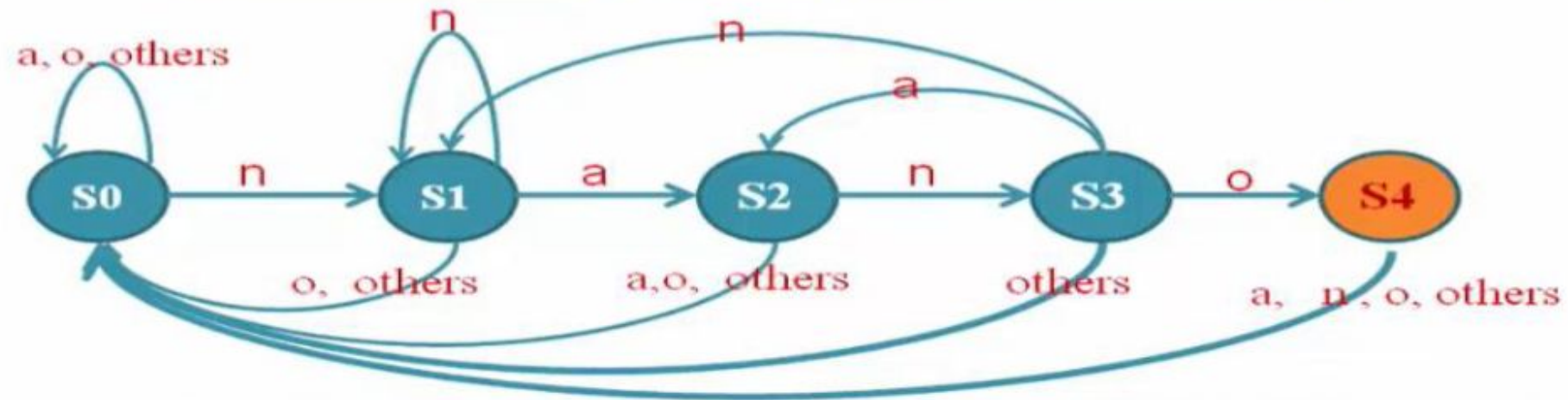
States	n	a	o	others
S0	S1	S0	S0	S0
S1	S1	S2	S0	S0
S2	S3	S0	S0	S0
S3	S1	S2	S4	S0
S4	S0	S0	S0	S0

Example Text = **b a n a n a n o n a**
 Pattern = **n a n o**

Transition Table

States	n	a	o	others
S0	S1	S0	S0	S0
S1	S1	S2	S0	S0
S2	S3	S0	S0	S0
S3	S1	S2	S4	S0
S4	S0	S0	S0	S0

Finite Automata



Text

b	a	n	a	n	a	n	o	n	a
---	---	---	---	---	---	---	---	---	---

S0

S0

S0

S1

S2

S3

S2

S3

S4